TU Wien

Business Informatics Group
Institut für Softwaretechnik und Interaktive Systeme

Web-Transaktionen in der Praxis

Evaluierung und Vergleich bestehender Ansätze und Technologien für die Verarbeitung von Single-Website und Website übergreifenden Transaktionen

Diplomarbeit zur Erlangung des akademischen Grades eines Magister rerum socialium oeconomicarumque

eingereicht bei o. Univ.-Prof. Mag. Dipl.-Ing. Dr. Gerti Kappel

Christian Schlosser

Wien, 29. September 2005

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Wien, 29. September 2005

Danksagung

Zuallererst bedanke ich mich herzlich bei meiner Betreuerin o. Univ.-Prof. Mag. Dipl.-Ing. Dr. Gerti Kappel für die Möglichkeit, die Diplomarbeit bei ihr schreiben zu können, die äußerst gute und kompetente Zusammenarbeit sowie die gegebenen Impulse und Anregungen bei der Verfassung dieser Arbeit.

Mein besonderer Dank gilt meiner Mutter Leopoldine Schlosser für all die Unterstützung und Entbehrungen, die notwendig waren, um mir dieses Studium und somit auch diese Diplomarbeit zu ermöglichen. Danke vielmals!

Auch bei meinem Bruder Michael Schlosser und seiner Partnerin, die mir während meines Studiums und der Diplomarbeit immer ein wichtiger Rückhalt waren, bedanke ich mich herzlich.

Bei Manuel Wimmer, Markus Moldaschl und Doris Meier, die mich beim Korrekturlesen der Arbeit unterstützt haben, bedanke ich mich ebenso, wie bei allen, die den Abschluss dieser Arbeit und des Studiums in irgendeiner Form positiv beeinflusst haben.

Kurzfassung

Die Entwicklung des Internet in den letzten Jahren und Jahrzehnten hat dazu geführt, dass die Anforderungen an angebotene Dienste und deren zugrunde liegende Applikationen ständig gestiegen sind und steigen. Um den Anforderungen gerecht werden zu können, ist oftmals eine Kooperation von mehreren, unabhängigen und heterogenen Systemen zur Bündelung mehrerer Dienste notwendig.

Da die Art und Weise der Ausführungen von Diensten von dem zu realisierenden, oftmals komplexen Geschäftsprozess abhängt, ist es sehr wichtig, die einzelnen Aktivitäten eines solchen Prozesses zu koordinieren, um ein gemeinsames Ergebnis unter den teilnehmenden Diensten zu erreichen. Weil die Abarbeitung eines Geschäftsprozesses oft länger dauern kann, ist es außerdem ganz wichtig, Nebenläufigkeit und damit verbesserte Performanz zu gewährleisten, um auch große Benutzerzahlen handhaben zu können. Diese Anforderungen werden im Rahmen von Transaktionsmodellen behandelt.

In dieser Diplomarbeit wird deshalb ein Überblick über relevante Teile der Transaktionstheorie gegeben. Das Augenmerk liegt dabei auf offenen geschachtelten Transaktionen, mittels deren Einsatz sehr komplexen Sachverhalten Rechnung getragen werden kann. Basierend auf der existierenden Transaktionstheorie ist die Untersuchung von drei Transaktionsprotokollen für Web Services zentraler Punkt dieser Arbeit. Alle drei ermöglichen es, mehrere Web Services untereinander zu koordinieren und ein gemeinsames Ergebnis herbeizuführen. Neben einer detaillierten Erklärung der einzelnen Protokolle werden diese hinsichtlich ihrer Eignung für die Abbildung langer Geschäftsprozesse kritisch betrachtet und gegenübergestellt.

Um den praktischen Einsatz von Transaktionsprotokollen für Web Services zu demonstrieren, wird neben der Vorstellung einiger aktueller Projekte und Produkte der durchgeführte Test einer Demo-Applikation und die dabei gewonnen Eindrücke, die durchaus sehr positiv waren, geschildert. Auch die in der Praxis sehr wichtigen Frameworks J2EE und .NET werden hinsichtlich ihrer Transaktionsmöglichkeiten untersucht und kritisch betrachtet.

Abstract

The evolution of the Internet has led to the fact that the requirements at services offered and their underlying applications constantly have been rising during the last years. In order to fulfil these requirements, co-operation of several independent and heterogeneous systems for bundling multiple services is necessary.

Since the interplay and execution of services has to reflect the underlying often complex business process, it is very important to coordinate the different activities of such a process to reach a common outcome among the participating services. In addition, business processes may take a long time, thus, ensuring concurrency and high performance is also very important, in order to be able to handle a large quantity of users. These requirements are all treated by transaction models.

In the course of this diploma thesis, an overview of relevant parts of the transaction theory is being given. Furthermore, the attention is drawn to open nested transactions, which are quite useful for complex circumstances. Based on the existing transaction theory, the investigation of three transaction protocols for Web Services, which make it possible to coordinate several Web Services in order to reach a common outcome, is the central point of this work. In addition to a detailed explanation of the protocols, their suitability for representing long business processes is extensively examined and then compared.

In order to give a demonstration of the practical use of transaction protocols for Web Services, the accomplished test of a demo application and the acquired impressions, which were above all positive, are described. Additionally, some current projects and products are introduced. Finally, the transaction possibilities of the very important frameworks in practice, J2EE and .NET, are analysed and verified.

Inhaltsverzeichnis

KAPITEL 1

EINLEITUNG 1

1.1 BEISPIEL EINER PROBLEMSTELLUNG 2

1.2 ZIEL DIESER DIPLOMARBEIT 3

1.3 AUFBAU DER ARBEIT 3

KAPITEL 2

TRANSAKTIONEN 5

2.1 BEGRIFFE UND PROBLEME 8

2.1.1 Dirty Read & Cascading Abort 8

2.1.2 Lost Update 9

2.1.3 Unrepeatable Read 10

2.1.4 Phantom Read 10

2.2 SERIALISIERUNG 11

2.2.1 Konfliktäquivalenz und –serialisierbarkeit 13

2.2.2 View-Äquivalenz und –Serialisierbarkeit 14

2.2.3 Serialisierungsgraphen 15

2.3 TWO-PHASE LOCKING 17

2.3.1 Deadlocks 18

2.4 ACID 19

2.5 TRANSAKTIONSARTEN 21

2.5.1 Verteilte Transaktionen 22

2.5.2 Geschachtelte Transaktionen 27

KAPITEL3

TRANSAKTIONSPROTOKOLLE UND FRAMEWORKS FÜR WEB SERVICES 32

3.1 WEB SERVICES UND TRANSAKTIONSMANAGEMENT ALLGEMEIN 32

3.1.1 SOAP 33

3.1.2 UDDI 34

3.1.3 WSDL 35

3.1.4 Transaktionsmanagement bei Web Services allgemein 35

3.2 OASIS BUSINESS TRANSACTION PROTOCOL 37

3.2.1 Transaktionsunterstützung 41

3.2.2 BTP und Holiday Package ... *44*
3.3 WS-COORDINATION UND WS-TRANSACTION ... 46
3.3.1 Web Services Coordination ... *46*
3.3.2 Web Services Atomic Transaction ... *49*
3.3.3 Web Services Business Activity Framework ... *52*
3.3.4 WS-Tx und Holiday Package ... *55*
3.4 WS-COMPOSITE APPLICATION FRAMEWORK ... 58
3.4.1 Zusammenhang zwischen WS-CTX, WS-CF und WS-TXM ... *58*
3.4.2 ACID Transactions ... *60*
3.4.3 Long Running Action ... *62*
3.4.4 Business process transaction ... *64*
3.4.5 WS-CAF und Holiday Package ... *70*
3.5 GEGENÜBERSTELLUNG DER PROTOKOLLE ... 73

KAPITEL 4

TRANSAKTIONEN IN J2EE UND .NET ... 76

4.1 TRANSAKTIONEN IN J2EE ... 76
4.1.1 Transaktionssteuerung ... *80*
4.1.2 Wichtiges zu Transaktionen mit J2EE ... *85*
4.1.3 JDBC ... *85*
4.1.4 J2EE und Holiday Package ... *86*
4.2 TRANSAKTIONEN IN .NET ... 87
4.2.1 Automatische Transaktionen ... *88*
4.2.2 Benutzerdefinierte Transaktionen ... *90*
4.2.3 .NET und Holiday Package ... *92*

KAPITEL 5

IMPLEMENTIERUNGEN VON TRANSAKTIONSPROTOKOLLEN ... 93

5.1 ARJUNA TRANSACTION SERVICE SUITE 4.0 ... 93
5.1.1 WS-C/WS-Tx Demo ... *94*
5.1.2 Fazit ... *97*
5.2 JOTM-BTP ... 97
5.3 APACHE KANDULA ... 98
5.4 WEITERE IMPLEMENTIERUNGEN ... 98

KAPITEL 6

ZUSAMMENFASSUNG UND AUSBLICK ... 99

ABBILDUNGSVERZEICHNIS ... 102

TABELLENVERZEICHNIS 104

LITERATURVERZEICHNIS 105

Kapitel 1

Einleitung

Das Internet und die darin angebotenen Dienste haben von der Zeit ihrer Entstehung bis heute eine starke Wandlung erlebt. Waren es anfangs nur statische Inhalte, die dem User präsentiert wurden und in weiterer Folge dynamische, datenbankgestützte Inhalte, die sich auf wenige Funktionen beschränkt haben, sind im Laufe der Zeit richtige Portale entstanden, die weit mehr zu bieten haben, als bloßes Registrieren für einen Newsletter oder das Erstellen eines Warenkorbs. Hinter diesen Portalen stecken sehr viele unterschiedliche Technologien, die (teilweise) inkompatibel zueinander sind, wodurch die gebotenen Leistungen auf einen kleinen Bereich beschränkt sind. Möchte man demnach Leistungen untereinander austauschen, so ist es notwendig, geeignete Schnittstellen zu definieren. Dieser Versuch, Kompatibilität zwischen verschiedenen, heterogenen Systemen herzustellen, wurde unter anderem mit der Entwicklung von Web Services unternommen. Mit Hilfe von Web Services ist es möglich, Dienste mit einer vordefinierten Schnittstelle jedermann zugänglich zu machen.

Das Nutzen verschiedener Dienste unterschiedlicher Anbieter ermöglicht es anderen Anbietern wiederum, diese Dienste zu einem Dienst zusammenzufassen und als eigenen Dienst anzubieten. Werden nun mehrere Dienste für ein und dieselbe Aufgaben-/Problemstellung herangezogen, ist es oft wichtig, eine Abstimmung und Koordination unter diesen Diensten zu erreichen, um Inkonsistenzen zu vermeiden und die gesamte Problemstellung erfolgreich lösen zu können und nicht nur Teile davon. Genau mit dieser Materie – der Abstimmung und Koordination mehrerer Komponen-

ten, um einen gemeinsamen, konsistenten Ausgang unter diesen zu erreichen - beschäftigt sich die Transaktionstheorie.

Die Frage ist nun einerseits, welche theoretischen Ansätze und Überlegungen für die – auf mehrere (heterogene) Systeme verteilten – Komponenten relevant sind und andererseits, ob und in welcher Form aktuelle Ansätze die Lösung eines derartigen Problems unterstützen und in wieweit auf theoretische Konzepte aufgebaut und diesen Rechnung getragen wird.

1.1 Beispiel einer Problemstellung

Als begleitendes Beispiel, anhand dessen die Möglichkeiten, der einzelnen in weiterer Folge untersuchten theoretischen Konzepte, Ansätze und Technologien ausgeleuchtet werden, dient das so genannte Holiday Package, das in dieser Arbeit als „Referenzproblemstellung“ herangezogen wird.

Die Problemstellung ist, dass ein Urlaubswilliger eine Urlaubsreise, die Flug, Hotelzimmer und Mietauto beinhaltet, buchen möchte – am bequemsten natürlich alles über eine Internetseite, vorzugsweise die eines Reisebüros.

Wichtig ist dem Urlauber selbstverständlich, dass entweder alle Teile gebucht werden, oder gar keiner. Nützt ja nichts, wenn zwar das Hotel gebucht wird, aber kein Flug möglich ist. Beim Mietauto ist dies freilich nicht so tragisch. Kurz gesagt: alle vom User ausgewählten und gewünschten Komponenten sollen entweder erfolgreich oder gar nicht gebucht werden.

Die einzelnen Komponenten Flug, Hotelzimmer und Mietauto können natürlich von verschiedenen Anbietern stammen und werden dem User nur auf einer Website bequem angeboten. Im Hintergrund können also mehrere (heterogene) Systeme zusammenspielen, was auch nahe liegend ist, da beispielsweise das Reisebüro kein direkter Anbieter von Flügen ist, sondern diese von einer Fluggesellschaft angeboten werden. Dies bedeutet gleichzeitig, dass z.B. angebotene Flugbuchungsdienste von verschiedenen Reisebüros genutzt werden können.

Weiterer Teil des Holiday Package Beispiels ist die Berücksichtigung des Userverhaltens. Hat der User eine verfügbare Komponente ausgewählt, soll diese dann auch bei Abschluss der Buchung für ihn verfügbar sein und nicht während des Auswahlprozesses von einem anderen User gebucht und quasi weggeschnappt werden kön-

nen. Dies ist ein ganz wichtiger Punkt, da die Art der Lösung dieses Problems sehr stark mit Performanz und Nebenläufigkeit korreliert ist.

Zur Berücksichtigung des Userverhaltens gehört auch, dass der User seine vorläufig getroffenen Entscheidungen eventuell wieder revidiert, etwa, wenn er einen billigeren Flug oder ein schöneres Hotel findet.

1.2 Ziel dieser Diplomarbeit

Nimmt man den Ausgangspunkt – das Zusammenspiel von (heterogenen) Systemen - und die Problemstellung des Holiday Packages her, liegt die Zieldefinition relativ klar auf der Hand. Diese ist, einen Überblick über derzeitige Ansätze und Möglichkeiten zu geben, die für die Lösung der Problemstellung relevant sind, um die Frage beantworten zu können: „Wenn ich Single-Website und/oder vor allem Website übergreifende Transaktionen verarbeiten möchte, welche Möglichkeiten habe ich, dieses Vorhaben umzusetzen?“.

1.3 Aufbau der Arbeit

Diese Diplomarbeit besteht im Wesentlichen aus drei Teilen. Während der erste Teil die theoretischen Grundlagen näher zu bringen versucht, werden im zweiten Teil aktuelle Protokolle und Frameworks, die sich mit der gegebenen Problemstellung auseinandersetzen, sowie wichtige Konzepte und Technologien im Bereich des Webs untersucht. Der dritte Teil widmet sich der praktischen Umsetzung von im zweiten Teil behandelten Lösungsansätzen.

Das zweite Kapitel behandelt die Transaktionstheorie. Neben wichtigen Grundlagen im Bereich von Transaktionen werden auch einige relevante Transaktionsmodelle vorgestellt. Kapitel 2 stellt zum Einen das theoretische Fundament dieser Arbeit dar und soll zum Anderen der LeserIn einen Einblick in die Transaktionstheorie geben und deren Wichtigkeit und Notwendigkeit übermitteln.

Kapitel 3 beschäftigt sich mit Web Services im Allgemeinen sowie derzeitigen Protokollen und Frameworks zur Durchführung von Transaktionen mit Web Services. Anschließend an die Identifizierung der Möglichkeiten und technischen Abläufe jedes der Protokolle beziehungsweise Framworks erfolgt eine kritische Betrachtung dieser im Hinblick auf die Lösung des Holiday Package Problems.

In Kapitel 4 werden die Transaktionsmöglichkeiten in J2EE und .NET – auch unter Rücksichtnahme auf das Holiday Package – untersucht.

Kapitel 5 stellt einige Produkte und Projekte vor, die die in Kapitel 3 und 4 vorgestellten Ansätze und Konzepte in konkreten Implementierungen umgesetzt haben oder sich mit der Umsetzung beschäftigen.

Im letzten Kapitel werden schlussendlich eine kurze Zusammenfassung der gewonnenen Erkenntnisse sowie ein kurzer Ausblick auf weitere interessante Untersuchungsgegenstände im Zusammenhang mit Web-Transaktionen gegeben.

Kapitel 2

Transaktionen

In diesem Kapitel werden wichtige Teile der Transaktionstheorie behandelt sowie einige für die weiteren Inhalte dieser Arbeit interessante Transaktionsformen vorgestellt. Die Inhalte, die in diesem Kapitel besprochenen werden, wurden [Bern87], [Elma92], [Elmas02], [Gray93], [Litt04] und [Öszu99] entnommen.

Transaktionen sind eine Reihe von teilweise geordneten, logisch miteinander verknüpfte Operationen, oder wie in [Bern87] formuliert: „Aus Benutzersicht ist eine Transaktion die Ausführung eines oder mehrerer Programme, die Datenbank- und Transaktionsoperationen beinhalten.“. Da es in dieser Arbeit allerdings nicht um die Datenbankebene geht, können Transaktionen auch jegliche andere Operationen auf höheren Abstraktionsebenen beinhalten (z.B. Use Cases eines Geschäftsprozesses). Unter einer Transaktion kann auch durchaus eine Menge verschiedener Aufgabenstellungen verstanden werden, die logisch miteinander verknüpft sind.

Das Zusammenfassen mehrerer Operationen zu einer Transaktion ist deshalb wichtig, da bei auftretenden Fehlern die Transaktion als ganzes davon betroffen ist und nicht nur die direkt von Fehlern betroffene(n) Operation(en). Zur Verdeutlichung dieses Umstandes ziehen wir das Beispiel einer elektronischen Überweisung über 100 € von Konto A auf Konto B heran.

Ohne die Berücksichtigung von Überziehungsrahmen usw. ergeben sich bei der Überweisung zwei Operationen:

1. Abbuchung von 100 € von Konto A
2. Gutschrift von 100 € auf Konto B

Ohne der Verwendung einer Transaktion werden die Operationen unabhängig voneinander ausgeführt. Tritt nun bei einer der beiden Operationen ein Fehler auf (Lesefehler, Schreibfehler, Übertragungsfehler, etc.), so ist die andere Operation davon nicht betroffen und wird ganz normal ausgeführt. Angewendet auf das Beispiel bedeutet das, dass beim Auftreten eines Fehlers in der zweiten Operation (Gutschrift auf Konto B) diese Gutschrift nicht erfolgt, das Geld jedoch von Konto A bereits abgebucht wurde, was natürlich nicht zielführend ist.

Werden die beiden Operationen hingegen im Kontext einer Transaktion ausgeführt, wirkt sich ein Fehler einer Operation auf alle anderen Operationen aus, indem die ganze Transaktion nicht durchgeführt wird. Bezogen auf das Beispiel würden Abbuchung und Gutschrift nur dann durchgeführt werden, wenn bei beiden Operationen keinerlei Fehler auftreten. Kann die Gutschrift nicht durchgeführt werden, wird die ganze Transaktion abgebrochen und alle bisherigen Operationen werden rückgängig gemacht (Rollback). Abbildung 2.1 versucht, den Unterschied von Verwendung und Nichtverwendung von Transaktionen noch einmal zu visualisieren.

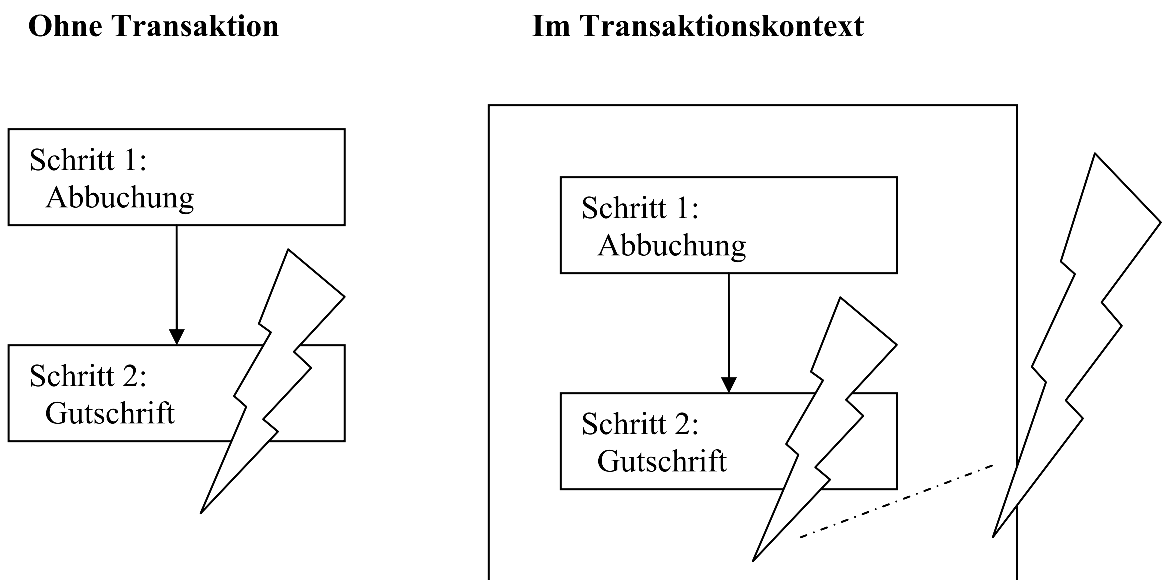


Abbildung 2.1: Ausführung von Operationen mit und ohne Transaktionskontext

Der Aufbau einer Transaktion sieht im Wesentlichen folgendermaßen aus:

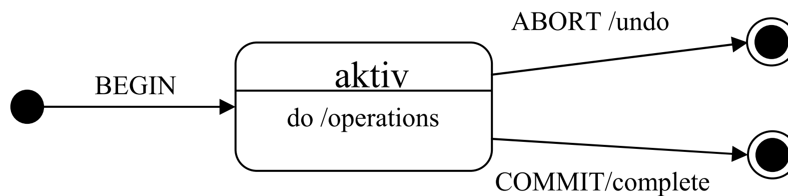


Abbildung 2.2: Aufbau einer Transaktion

Wie in Abbildung 2.2 dargestellt, gibt es beim Abschluss einer Transaktion die Möglichkeiten Commit und Abort. Eine Commit Operation impliziert, dass die Transaktion erfolgreich war und dass alle durch die Operationen entstandenen Effekte dauerhaft gemacht werden. Bei einem Abbruch (Abort) hingegen, der durchgeführt wird, wenn keine korrekte Ausführung der Operation möglich ist (beispielsweise durch Schreibfehler oder nicht erfüllte Bedingungen wie das unerlaubte Überziehen eines Kontoüberziehungsrahmens), werden alle durch die Transaktion entstandenen Effekte wieder aufgehoben bzw. rückgängig gemacht (Rollback). Eine abgebrochene Transaktion hat also de facto keinerlei Effekte.

Wurde eine Transaktion weder mit Commit, noch mit Abort beendet, so wird sie als aktiv bezeichnet. Solange eine Transaktion aktiv ist, kann nicht gesagt werden, ob sie letztendlich durchführbar ist oder nicht.

Da in den nächsten Unterkapiteln und Abschnitten genauer auf Transaktionsarten, Probleme und wichtige Begriffe eingegangen wird, wird an dieser Stelle eine Notation zur Darstellung von Transaktionen eingeführt, wie sie auch in [Bern87] verwendet wird. Als Operationen werden Lese- und Schreiboperationen herangezogen. Diese werden durch „lesen(<Feld>)“ und „schreiben(<Feld>,<Wert>)“ dargestellt. <Feld> bezeichnet das zu lesende bzw. zu (be)schreibende Feld und <Wert> den Wert, der in das Feld geschrieben wird. „lesen(x)“ bedeutet also, dass der Wert x gelesen, „schreiben(x,100)“ dass der Wert 100 ins Feld x geschrieben werden soll. Commit und Abort werden durch die jeweiligen Worte dargestellt. Der Beginn einer Transaktion wird in weiterer Folge nicht explizit angeschrieben. Die einzelnen Operationen, also `lesen`, `schreiben`, `commit` und `abort` werden jeweils durch „;“ getrennt.

Geht man davon aus, dass beim oben dargestellten Überweisungsbeispiel vor dem Schreiben der neuen Kontostände diese gelesen werden müssen, um den neuen Kon-

tostand zu berechnen und diesen dann zurückzuschreiben, könnte die Transaktion in der eingeführten Notation bei Anfangskontoständen von 3000 € folgendermaßen aussehen:

```
lesen(a);schreiben(a,2900);lesen(b);schreiben(b,3100);commit;
```

Da nun der grundsätzliche Aufbau von Transaktionen geklärt ist, werden im nächsten Unterkapitel einige wichtige Begriffe bzw. Probleme, die im Umgang mit Transaktionen auftreten können, vorgestellt.

2.1 Begriffe und Probleme

Bisher wurde nur eine Transaktion beleuchtet, doch die nun vorgestellten Probleme treten durchwegs bei gleichzeitiger (paralleler) Ausführung mehrerer Transaktionen auf. Zur besseren Illustrierung der Probleme und Begriffe wird die eben vorgestellte Notation verwendet. Zur Unterscheidung von zwei Transaktionen T_1 und T_2 werden die einzelnen Operationen durch Tiefgestellte Zahlen (Nummern der Transaktionen) erweitert. $Lesen_1(x)$ heißt demnach, dass Transaktion T_1 den Wert des Feldes x liest (lesen möchte).

2.1.1 Dirty Read & Cascading Abort

Dirty Read bedeutet, dass eine Transaktion T_2 einen Wert liest, der von einer anderen Transaktion T_1 verändert, die Änderung allerdings noch nicht committed wurde und die Änderung für T_2 sichtbar ist. Also:

$schreiben_1(x,2);lesen_2(x);schreiben_2(y,3);$[1]

Soweit scheint es, als wäre dieser Vorgang unproblematisch, was auch stimmt, wenn Transaktion T_1 mit „commit;“ abgeschlossen wird:

$schreiben_1(x,2);lesen_2(x);schreiben_2(y,3);commit_1;$

Wird T_1 allerdings mit „abort;“ beendet, also abgebrochen,

$schreiben_1(x,2);lesen_2(x);schreiben_2(y,3);abort_1;$

[1] Es wird davon ausgegangen, dass T_2 den Wert von x zur Berechnung von y benötigt.

wird T_1 und damit das Schreiben in das Feld x durch T_1 rückgängig gemacht. Das bedeutet allerdings für Transaktion T_2, dass diese einen Wert von x gelesen hat, der keine Gültigkeit mehr besitzt. Deshalb muss auch T_2 rückgängig gemacht werden. Bei diesem Vorgang spricht man von „Cascading Abort". Verhindert kann ein Cascading Abort dadurch werden, dass Transaktionen nur Werte lesen, die bereits committed wurden.

Oft reicht das Verhindern von Cascading Aborts allerdings nicht aus, da es durch die Technik des Wiederherstellens des vorherigen Images (restoring before image; zurückkehren auf einen früheren (Datenbank-)Status) bei Aborts zu weiteren Problemen kommen kann. Deshalb sind bei der Ausführung oft weitere Restriktionen notwendig (Stichwort: Strict Execution). Für eine genaue Beschreibung wird die LeserIn auf [Bern87] verwiesen.

Dirty Reads werden typischerweise zugelassen, um die Performanz zu erhöhen bzw. wenn man weiß, dass gleichzeitiger Zugriff nicht auftreten kann. [Litt04]

2.1.2 Lost Update

Zur Illustrierung des Lost Update Problems stelle man sich folgende Situation vor: Person 1 und Person 2 möchten in etwa zeitgleich 150 € bzw. 50 € auf Konto x (Kontostand davor ist 50 €) einzahlen. Mathematisch gesehen müsste der Kontostand nach den beiden Einzahlungen insgesamt 250 € betragen. Da beide Personen beispielsweise ein Programm, das den alten Betrag liest, den neuen Betrag berechnet und auf das Konto schreibt, ausführen, könnten die einzelnen Transaktionen isoliert von einander betrachtet wie folgt aussehen:

```
lesen1(x);schreiben1(x, 50 + 150);commit1;  bzw.
lesen2(x);schreiben2(x, 50 + 50);commit2;
```

Beide Transaktionen sind für sich genommen korrekt und ergeben einen neuen Kontostand von 200 € bzw. 100 €. Durch das gleichzeitige Ausführen der Programme kann es allerdings zu folgender Konstellation kommen (Anm.: die verschiedenen Konstellationen werden in weiterer Folge auch *Ausführungsplan* genannt):

```
lesen1(x);lesen2(x);schreiben1(x, 50 + 150);
commit1;schreiben2(x, 50 + 50);commit2;
```

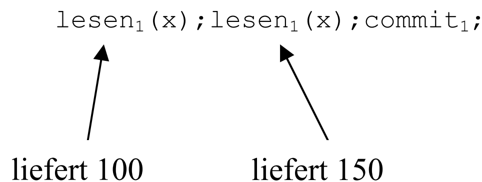
Wie leicht zu erkennen ist, wird der neu von Transaktion T1 geschriebene Kontostand durch den neuen von T2 überschrieben. Der neue Kontostand nach Durchfüh-

rung von T1 und T2 beträgt als nicht 250 € sondern nur 100 €. Die 150 € von T1 sind verloren gegangen.

Grund für das Verlorengehen ist das Lesen des gleichen (alten) Kontostandes von T1 und T2. [Bern87] definiert das Lost Update Problem sehr gut: „Das Lost Update Phänomen tritt auf, wenn zwei Transaktionen versuchen, einen Datensatz zu ändern und beide den alten Datensatz lesen, bevor ein der beiden den neuen Wert schreibt".

2.1.3 Unrepeatable Read

Von einem Unrepeatable Read (nicht wiederholbarem Lesezugriff) spricht man, wenn innerhalb einer Transaktion T_1 das mehrmalige Lesen eines Feldes verschiedene Werte liefert:



Hervorgerufen werden kann dieses Phänomen durch das Schreiben und committen des Feldes einer anderen Transaktion T_2 während der beiden Lesevorgänge von T_1.

```
lesen1(x);schreiben2(x,150);commit2;lesen1(x);commit1;
```

Der Unterschied zwischen dem Unrepeatable Read und dem Dirty Read ist der, dass Unrepeatable Read auch auftreten kann, wenn nur commitete Werte gelesen werden (siehe Beispiel), währenddessen das Dirty Read Problem damit gelöst werden kann.

2.1.4 Phantom Read

Phantom Read oder auch Phantomproblem genannt, entsteht dann, wenn während der Durchführung einer Transaktion neue Werte (Datensätze) von anderen Transaktionen eingefügt werden. Zum besseren Verständnis der Problematik soll folgendes Beispiel (nach [Bern87]) dienen.

Gegeben sind zwei Tabellen, die die Werte der Bestände einer Bank darstellen. Eine mit den Konten und eine mit den Gesamtsalden der Filialen (Abbildung 2.3). Man stelle sich nun vor, dass die Bank eine Kontrollapplikation hat, die alle Kontostände einer Filiale mit dem Saldo vergleicht, um Fehler zu entdecken. Das Programm liest

also beispielsweise alle Konten der Filiale A ein, zählt diese zusammen, liest dann den Saldo der Filiale ein und vergleicht diesen mit den aggregierten Kontobeträgen.

Kontentabelle:

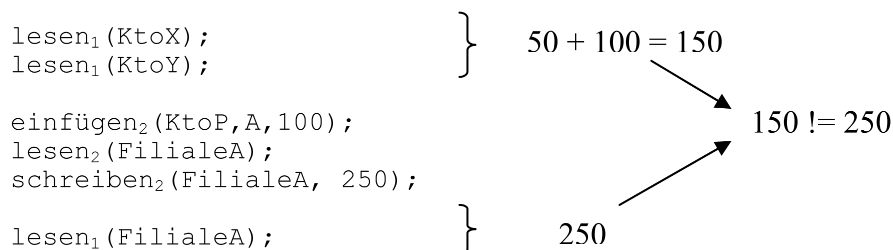
Kto	Filiale	Betrag
KtoX	A	50
KtoY	A	100
KtoZ	B	300
KtoP	*A*	*100*

Saldentabelle:

Filiale	Saldo	
Filiale A	150	→ *250*
Filiale B	300	

Abbildung 2.3: Konten einer Bank (nach [Bern87])

Wird nun zwischen dem Lesen der Einzelkonten und dem Lesen des Saldos ein neuer Datensatz von einer anderen Transaktion eingefügt (in Abbildung 2.3 kursiv und im Ausführungsplan durch „einfügen(<Feld>,<Filiale>,<Wert>)“ dargestellt), ändert sich natürlich der Saldo der Filiale und die Kontrollapplikation kommt zu einem inkonsistenten Ergebnis.



KtoP ist also wie eine Art Phantom, das auftaucht und wieder verschwindet.

2.2 Serialisierung

Wie nur unschwer zu erkennen ist, entstehen die im letzten Unterkapitel vorgestellten Probleme dadurch, weil verschiedene Transaktionen überlappt ausgeführt werden. Um diese Probleme in den Griff zu bekommen, muss man einen Mechanismus finden, um die Überlappungen zu kontrollieren.

Die einfachste Möglichkeit wäre, alle Transaktionen hintereinander, also seriell auszuführen. Zuerst werden also alle Operationen von T_i ausgeführt und dann alle Operationen von T_j. Da davon ausgegangen wird, dass die einzelnen Transaktionen korrekt sind, ist auch eine serielle Ausführung dieser korrekt. Diese Art der Ausführung

würde zwar die Korrektheit garantieren, doch würden CPU-Kapazitäten nicht optimal genutzt beziehungsweise bei länger dauernden Transaktionen das ganze System blockiert. Die sehr starke Einschränkung der Nebenläufigkeit ist allerdings nicht notwendig, da zwei Transaktionen, die verschiedene Daten verarbeiten ohne weiteres nebeneinander ablaufen können.

Die überlappende Ausführung von Transaktionen ist also zulässig, wenn diese Ausführung äquivalent zu einer seriellen Ausführung ist. Man nennt diese Ausführung dann serialisierbar.

> *Definition:* Ein Ausführungsplan S mit n Transaktionen ist serialisierbar, wenn er zu einem seriellen Ausführungsplan mit den gleichen n Transaktionen äquivalent ist [Elma02].

Der Ausführungsplan

$lesen_1(x);lesen_2(y);schreiben_2(y)$[2]$;commit_2;schreiben_1(x);commit_1;$

ist also äquivalent zur seriellen Ausführung $T_1;T_2;$

$lesen_1(x);schreiben_1(x);commit_1;lesen_2(y);schreiben_2(y);commit_2;$

beziehungsweise $T_2;T_1;$

$lesen_2(y);schreiben_2(y);commit_2;lesen_1(x);schreiben_1(x);commit_1;$

Der Ausführungsplan führt, wie leicht nachvollziehbar ist, zum selben Ergebnis wie die serielle Ausführung von T_1 und T_2 (in jeglicher Reihenfolge). Er ist also äquivalent zu (mindestens) einem seriellen Ausführungsplan und somit serialisierbar.

Nachdem Serialisierbarkeit bedeutet, dass das Ergebnis das gleiche wie bei einer seriellen Ausführung ist, ist sie eine Definition von Korrektheit. Nimmt man das Lost Update- und das Unrepeatable Read Problem her, sieht man, dass diese beiden nicht serialisierbar sind, jedoch können diese bei Verwendung der Serialisierbarkeit per Definition nicht auftreten.

Für die Äquivalenz von Ausführungsplänen werden allgemein zwei Definitionen verwendet: Die Konfliktäquivalenz und die View-Äquivalenz

---

[2] Werte werden vernachlässigt, weil hier nicht relevant

2.2.1 Konfliktäquivalenz und –serialisierbarkeit

Um Konfliktäquivalenz beschreiben zu können, muss zuerst der Begriff Konflikt definiert werden: Ein Konflikt zwischen zwei Operationen liegt dann vor, wenn die Reihenfolge der Operationen Einfluss auf deren Ergebnis hat.

Wir unterscheiden fünf Möglichkeiten:

- Zugriff auf verschiedene Objekte
  Greifen zwei Operationen auf verschiedene Objekte zu (lesend oder schreibend), ist es egal, in welcher Reihenfolge sie das tun, das Ergebnis wird das selbe sein.
- Lesen / Lesen:
  Lesen zwei Operationen vom selben Objekt, so ist die Reihenfolge – wie beim Zugriff auf verschiedene Objekte – egal.
- Lesen / Schreiben:
  Führt eine Operation einen Lese- und eine andere danach einen Schreibzugriff auf das selbe Objekt aus, ist die Reihenfolge entscheidend. Würde man die Reihenfolge ändern, also zuerst schreiben und dann lesen, wäre das Ergebnis potenziell ein anderes. Lese- und Schreiboperationen stehen aus diesem Grund miteinander in Konflikt
- Schreiben / Lesen
  Analog zu Lesen / Schreiben
- Schreiben / Schreiben
  Wie schon bei den beiden vorigen Konstellationen ist bei zwei auf das selbe Objekt schreibend zugreifenden Operationen die Reihenfolge von Bedeutung:
  `schreiben1(x,100);schreiben2(x,50);   neq`[3] `schreiben2(x,50);schreiben1(x,100);`
  Auch diese beiden Operationen stehen miteinander in Konflikt.

Zwei Operationen stehen also miteinander in Konflikt, wenn beide auf das selbe Objekt zugreifen und zumindest eine der beiden eine Schreiboperation ist [Elma92]. In Abbildung 2.4 ist dieser Sachverhalt noch einmal dargestellt.

---

[3] not equal (=ungleich)

	lesen	schreiben	
	Nein	Ja	lesen
	Ja	Ja	schreiben

Abbildung 2.4: Konfliktrelationen zwischen Lese- und Schreiboperationen (nach [Elma92])

Da der Begriff Konflikt geklärt wurde, nun zur Konfliktäquivalenz: Ein Ausführungsplan ist zu einem seriellen Ausführungsplan konfliktäquivalent, wenn die Reihenfolge der in Konflikt stehenden Operationen in beiden Ausführungsplänen gleich ist. Der Ausführungsplan ist damit konfliktserialisierbar.

Folgender Ausführungsplan (nach [Elma92]) ist konfliktserialisierbar:

$Schreiben_1(x);lesen_2(x);commit_2;schreiben_3(y);$
$commit_3;schreiben_1(y);commit_1;$

Dieser Plan ist konfliktäquivalent zur seriellen Ausführung $T_3;T_1;T_2$, da y jeweils von T_3 geschrieben und dann von T_1 geschrieben beziehungsweise x jeweils von T_1 geschrieben und dann von T_2 gelesen wird.

2.2.2 View-Äquivalenz und –Serialisierbarkeit

View-Äquivalenz zwischen einem Ausführungsplan und einem seriellen Ausführungsplan liegt dann vor, wenn

- beide die gleiche Transaktionsmenge beinhalten sodass
- in beiden Ausführungsplänen jede Transaktion die gleichen Werte liest und
- die Endzustände der Datenbankobjekte die gleichen sind

Die Leseoperationen müssen also in beiden Ausführungsplänen die gleichen Werte lesen (haben also die gleiche View) und die Endzustände der Objekte müssen gleich sein.

Wenn diese Bedingungen erfüllt sind, sind die beiden Ausführungspläne View-äquivalent und damit View-serialisierbar. Folgendes Beispiel (nach [Elma92]) zeigt einen View-serialisierbaren Ausführungsplan:

$schreiben_1(x);schreiben_2(x);schreiben_3(x);$

```
schreiben2(y);lesen1(y);
```

ist View-äquivalent zur seriellen Ausführung $T_2;T_1;T_3$ weil in beiden Fällen T_1 von T_2 geschriebenen Wert von y liest und der endgültige Wert von x durch T_3 beziehungsweise der von y durch T_2 geschrieben wird.

Der soeben dargestellte Ausführungsplan ist zwar View-serialisierbar, jedoch nicht konfliktserialisierbar, da ja die Reihenfolge aller in Konflikt stehender Operationen bei dieser Form der Serialisierbarkeit gleich sein muss. Da aber $schreiben_1(x)$ vor $schreiben_2(x)$ und $schreiben_2(x)$ vor $lesen_1(x)$ kommen müsste, kann kein serieller Ausführungsplan gefunden werden, der diese Bedingung erfüllt.

Wie in Abbildung 2.5 dargestellt, ist die Konfliktserialisierbarkeit stärker. Wenn ein Ausführungsplan konfliktserialisierbar ist, ist er auch View-serialisierbar. Umgekehrt ist das normalerweise nicht der Fall. Die in Abbildung 2.5 eingezeichnete „order preserving conflict serialisability“ kann in [Elma92] nachgelesen werden.

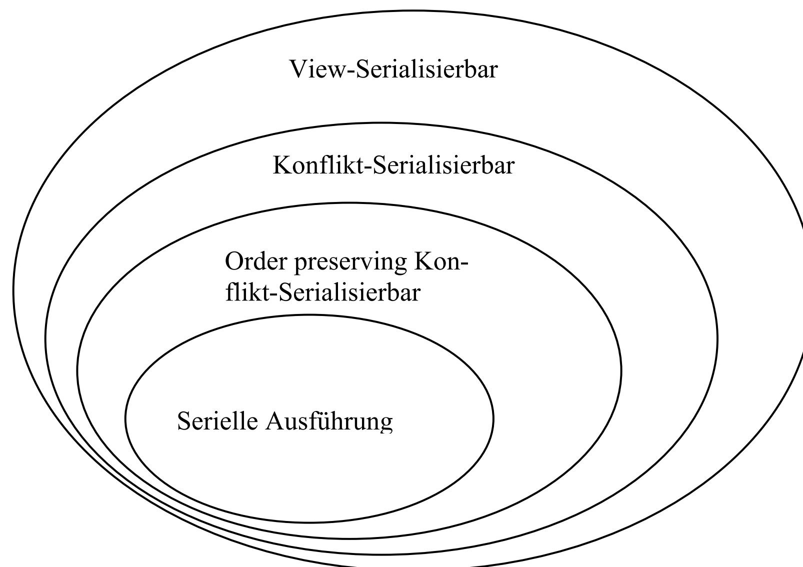


Abbildung 2.5: Zusammenhang zwischen Serialisierungstypen

2.2.3 Serialisierungsgraphen

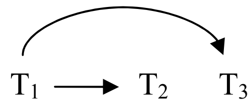
Um herauszufinden, ob ein Ausführungsplan konfliktserialisierbar ist, genügt es, einen vom Ausführungsplan abgeleiteten, so genannten Serialisierungsgraphen zu analysieren. Die Knoten des Graphen sind die Transaktionen. Eine Kante von Kno-

ten Ti nach Tj stellt einen Konflikt zwischen einer Operation von Ti und Tj dar. Ein Ausführungsplan ist dann konfliktserialisierbar, wenn er keine Zyklen enthält.

Bsp1:

```
schreiben₁(x);lesen₂(x);lesen₃(x);
```

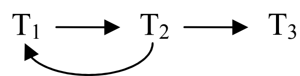
Graph:



Bsp2:

```
schreiben₁(x);schreiben₂(x);schreiben₃(x);
schreiben₂(y);lesen₁(y);
```

Graph:



Während Bsp1 konfliktserialisierbar ist (äquivalent zu T_1;T_2;T_3 bzw. T_1;T_3;T_2) enthält das bereits bei der View-Äquivalenz herangezogene Bsp2 einen Zyklus und ist daher nicht konfliktserialisierbar.

Wollte man nun in der Praxis einen Ausführungsplan mittels Serialisierungsgraphen auf Serialisierbarkeit testen, könnte dies erst passieren, nachdem der Ausführungsplan bekannt ist und ein negativer Test würde die Annullierung des Plans bedeuten. Da dies sehr problematisch ist, werden verschiedene Protokolle (Regeln) eingesetzt, die die Serialisierbarkeit von Ausführungsplänen sicherstellen.

Eines dieser Protokolle ist der Two-Phase-Locking Mechanismus, der im nächsten Unterkapitel genauer besprochen wird.

Da ein genauerer Einblick in die Theorie der Serialisierungsgraphen in dieser Arbeit nicht zielführend wäre, wird der interessierte Leser auf das Kapitel „Serialisability Theory“ in [Bern87] verwiesen.

2.3 Two-Phase Locking

Wie bereits erwähnt, dient das Two-Phase Locking Protokoll zur Sicherstellung der Serialisierbarkeit.

Two-Phase Locking funktioniert grundsätzlich so, dass Einträge, auf die von einer Transaktion lesend oder schreibend zugegriffen wird vor dem Zugriff gesperrt und nach dem Zugriff die Sperre wieder aufgehoben wird, um sicherzustellen, dass immer nur eine Transaktion auf einen Datensatz zugreift. Bei den Sperren selbst unterscheidet man Lese- und Schreibsperren für die jeweiligen Zugriffsarten.

Selbstverständlich können zwei nicht in Konflikt stehende Operationen gleichzeitig Einträge sperren. Die Sperren stehen also nicht miteinander in Konflikt. Stehen zwei Operationen miteinander in Konflikt, gehören aber zur gleichen Transaktion, so stehen die Sperren auch nicht in Konflikt, beide Operationen können also gleichzeitig sperren.

Das grundlegende Two-Phase Locking besteht aus drei Regeln:

1. Wenn eine Transaktion eine Sperre für eine Operation beantragt, wird überprüft, ob diese in Konflikt zu einer bestehenden Sperre einer anderen Transaktion ist. Ist dies der Fall, muss die Operation warten, bis die Sperre wieder freigegeben wird. Anderenfalls kann die Sperre gesetzt werden.
2. Die Sperre darf solange nicht freigegeben werden, bis die Operation verarbeitet ist.
3. Wenn eine Transaktion eine Sperre freigegeben hat, darf diese Transaktion keine weitere Sperre mehr beantragen.

Die dritte Regel wird auch die *Two Phase Regel* genannt und garantiert die Serialisierbarkeit. Da nur Sperren beantragt werden dürfen, solange noch keine Sperren freigegeben wurden, kann der Sperrmechanismus in Wachstums- und Schrumpfungs-Phase (growing- und shrinking-phase) eingeteilt werden. Daher der Name Two Phase Regel beziehungsweise Two Phase Locking. In der Wachstumsphase werden die Sperren nach und nach beantragt, die Anzahl der Sperren wächst also stetig und in der Schrumpfungsphase werden die Sperren (eigentlich gleichzeitig) freigegeben, die Anzahl der Sperren schrumpft. Der Verlauf der gehaltenen Sperren einer Transaktion über die Zeit ist in Abbildung 2.6 dargestellt.

Leider hat das Two-Phase Locking einen großen Nachteil: Es können Deadlocks entstehen.

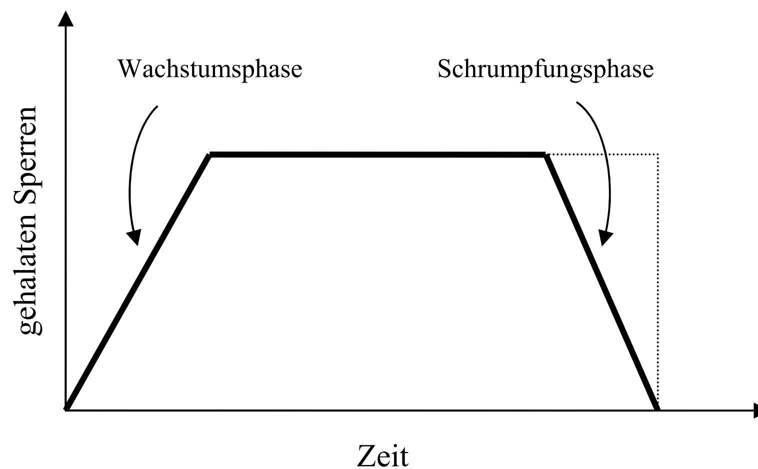


Abbildung 2.6: Two-Phase Locking (nach [Litt04])

2.3.1 Deadlocks

Wie bereits erwähnt, muss eine Transaktion $T_{i,}$ die eine Sperre für einen bereits von einer anderen Transaktion T_j gesperrten Datensatz beantragt, warten. Nun kann es aber passieren, dass T_j ebenfalls eine Sperre benötigt, die von T_i gehalten wird. Es muss also T_i darauf warten, dass T_j seine Sperren freigibt und vice versa. Diese Situation, die kein „natürliches" Ende hat, nennt man Deadlock, weil T_i und T_j sich gegenseitig blockieren. Das folgende Beispiel (nach [Bern87) soll diese unangenehme Situation visualisieren:

```
T1: lesen1(x);schreiben1(y);commit1;
T2: schreiben2(y);schreiben2(x);commit2;
```

Der Ablauf könnte nun folgendermaßen aussehen:

- T_1 sperrt den Datensatz x zum Lesen
- T_2 sperrt den Datensatz y zum Schreiben
- T_2 beantragt Sperre zum Schreiben von x, muss allerdings warten bis T_1 diese freigibt
- T_1 beantragt Sperre zum Schreiben von y und muss auf die Freigabe von T_2 warten

Da Deadlocks zu ungewollten, unendlichen Blockaden von Transaktionen führen, ist es wichtig, diese aufzuspüren. Dazu werden häufig zwei Techniken angewandt:

- Timeouts: Wenn eine Transaktion länger als eine vorgegebene Zeit auf eine Sperre wartet, wird davon ausgegangen, dass ein Deadlock vorliegt und die Transaktion wird daraufhin abgebrochen
- Erkennung durch Graphen: Ähnlich wie beim Serialisierungsgraphen werden die Abhängigkeiten der Sperren von Transaktionen mittels eines so genannten Waits-For-Graph dargestellt. Es gibt eine Kante von Knoten T_i zu Knoten T_j, falls T_i auf die Freigabe von Sperren der Transaktion T_j wartet ([Bern87]). Enthält der Graph Zyklen, liegt ein Deadlock (bzw. mehrere Deadlocks) vor und eine der betroffenen Transaktionen wird beendet.

Bei den Timeouts ist es wichtig, die Zeit richtig zu wählen. Wird die Zeit zu kurz bemessen, werden eventuell Transaktionen abgebrochen, die einfach nur länger dauern, aber in keinen Deadlock verwickelt sind. Wird die Zeit zu lange bemessen, kommt es beim Auftreten von Deadlocks zu unnötigen Verzögerungen.

Ähnliches gilt bei den Graphen: Werden die Graphen zu oft auf Zyklen untersucht, ist dies mit sehr hohem Aufwand verbunden, werden sie zu wenig oft untersucht, kann es zu unnötigen Blockaden kommen. Ein geeignetes Mittelmaß ist in beiden Fällen schwer zu finden und hängt sehr stark von den durchzuführenden Transaktionen ab.

Neben der bisher genannten Technik gibt es natürlich auch noch weitere, so zum Beispiel das konservative 2PL bei dem eine Transaktion alle in der Transaktion benötigten Sperren am Beginn beantragt und somit Deadlocks nicht vorkommen können. Nachteil: Die Nebenläufigkeit wird dadurch eingeschränkt.

Details über diese Technik, das strikte 2PL, das Index Locking, mit dem das Phantomproblem verhindert werden kann sowie die Möglichkeit der Granularität von Sperren (Sperren von Datensätzen, Tabellen, Datenbanken) und deren Auswirkungen auf die Performance werden hier nicht näher behandelt. Auch die Beschreibung von weiteren Protokollen zur Gewährleistung der Serialisierbarkeit, wie etwa das Timestamping wäre nicht zielführend. Der interessierte Leser wird deshalb auf die entsprechenden Kapitel in [Bern87] verwiesen.

2.4 ACID

Von transaktionsverarbeitenden Systemen werden häufig vier grundlegende Eigenschaften verlangt, die unter dem Akronym ACID zusammengefasst werden. ACID steht für Atomicity, Consistency, Isolation und Durability.

Atomicity:

Atomizität im Zusammenhang mit Transaktionen bedeutet, dass eine Transaktion entweder vollständig oder überhaupt nicht durchgeführt wird. Man spricht daher oft von der „Alles-oder-Nichts“ Eigenschaft. Eine Transaktion wird als atomar bezeichnet, wenn sie diese Eigenschaft erfüllt.
Eine Transaktion darf aus Benutzersicht – erfüllt sie die Atomizität – nur vom Ausgangspunkt zum Endpunkt „springen“, ohne Zwischenzustände zu erreichen. Wenn wir uns an das Überweisungsbeispiel am Beginn erinnern, so bedeutet Atomizität, dass entweder alle Teile der Überweisung oder gar kein Teil durchgeführt werden. Wurde die Abbuchung systemintern beispielsweise bereits durchgeführt und wird die Transaktion dann abgebrochen, so darf sie für den Benutzer nicht sichtbar sein und all ihre Wirkungen müssen rückgängig gemacht werden. Für Benutzer darf auch nicht sichtbar sein, dass die Transaktion abgebrochen wurde. „Atomar zu sein bedeutet, dass eine fehlgeschlagene Transaktion per Definition nichts gemacht hat…“ [Gray93] Sie hat offiziell also nie existiert.
Atomizität wird sehr oft durch das Two-Phase Commit Protokoll sichergestellt. Eine genaue Beschreibung dieses wird im Rahmen der verteilten Transaktionen (Abschnitt 2.5.1) besprochen.

Consistency:

Diese Eigenschaft verlangt, dass nach vollständiger Durchführung einer Transaktion die Konsistenz des Systems erhalten bleiben muss. Voraussetzung dafür ist, dass das System vor der Transaktion in einem konsistenten Zustand war und durch die Transaktion sozusagen von einem in einen anderen konsistenten Zustand übergeführt werden muss.
Konsistenz bedeutet, dass der Status der Datenbank alle Konsistenz- und Integritätsbedingungen der Applikation erfüllt. Die Konsistenzbedingung beim Überweisungsbeispiel wäre etwa, dass gleich von Konto A abgebucht wird, wie dem Konto B gutgeschrieben wird, oder die Bedingung, dass die Konten nicht überzogen werden dürfen.
Konsistenzbedingungen müssen von der ProgrammiererIn definiert und sichergestellt werden.

Isolation:

Isolation bedeutet, dass sich Operationen einer Transaktion bzw. die Transaktion als Ganzes so verhalten müssen(muss), als würde(n) sie alleine, isoliert von anderen Transaktionen ausgeführt. Dies entspricht genau der Eigenschaft

der Serialisierbarkeit, die ja bereits in Unterkapitel 2.2 eingehend besprochen wurde.
Bei der Eigenschaft der Isolation werden wiederum verschiedene Stufen unterschieden. So werden bei einer niedrigeren Isolationsstufe etwa Dirty-Read Operationen und das Lost-Update-Problem zugelassen, um die Nebenläufigkeit zu erhöhen. Natürlich setzt dies eine niedrige Konfliktrate voraus und ist sehr stark von der Art der durchgeführten Transaktionen abhängig. Bei hohen Isolationsstufen wird das Auftreten diverser Nebenläufigkeitsprobleme durch Sperren (oder ähnliches, siehe Abschnitt über Locking) von verwendeten Ressourcen verhindert.

Durability:

Durability bedeutet, dass Effekte einer mit Commit beendeten Transaktion dauerhaft gemacht werden muss beziehungsweise ihre Dauerhaftigkeit garantiert werden muss (natürlich kann es keine 100%ige Garantie geben (Plattencrash, Naturkatastrophen, etc.). Effekte von Transaktionen müssen also derart gesichert werden, dass beim Auftreten von Fehlern nach dem Commit diese jederzeit wiederhergestellt werden können. Mechanismen, die dies garantieren, können in [Elma02] und [Bern87] nachgelesen werden. (Stichwort: Recovery, Undo/Redo)

2.5 Transaktionsarten

Die einfachste Form von Transaktionen sind die bereits implizit in den Beispielen dieses Kaptitels verwendeten flachen oder Top-Level-Transaktionen. Flache Transaktionen werden deshalb so genannt, weil es nur eine Kontrollebene gibt. Alle Operationen zwischen dem Beginn und dem Commit befinden sich am gleichen Level – am Top-Level.

Diese Art der Transaktionen reicht zwar in den meisten Fällen aus, doch werden zur Ausführung von Transaktionen, wie sie das in der Einleitung vorgestellte Holiday Package Beispiel repräsentiert, Erweiterungen notwendig, um den Anforderungen solcher Transaktionen entsprechen zu können.

Bisher wurde im Laufe dieses Kapitels nicht explizit angenommen, dass die verschiedenen im Transaktionskontext durchgeführten Operationen auf mehrere Systeme aufgeteilt sind. Wie im Holiday Package Beispiel allerdings angenommen, kann es gut sein, dass der Flugbuchungsmechanismus sich auf einem anderen System be-

findet wie der Hotelbuchungsmechanismus. Um diese Koordination mehrerer verteilter Operationen realisieren zu können, gibt es das Konzept der *Verteilten Transaktionen*, das genau diesen Umstand behandelt und einen möglichen Lösungsansatz für das Holiday Package Problem darstellt.

Wie bereits erwähnt, reicht das Konzept der flachen Transaktionen zwar in den meisten Fällen aus, doch besitzt es viele Limitationen, die für die Ausführung des Holiday Package Beispiels aus praktischer Sicht nur schwer in Kauf genommen werden können. Flache Transaktionen bieten etwa keine Möglichkeit, die Aufteilung von Operationen in mehrere Schritte vorzunehmen. Tritt ein Fehler auf, oder muss eine Operation abgebrochen werden, bedeutet dies, dass Operationen, die bereits korrekt ausgeführt werden hätten können ebenfalls abgebrochen werden müssten. Wurden beispielsweise schon ein passendes freies Hotelzimmer und ein passendes Mietauto gefunden, wäre es wenig sinnvoll, diese wieder zu verwerfen, nur weil es Probleme bei der Flugbuchung gibt.

Ein weiterer Nachteil von flachen Transaktionen und ihrer fehlenden Möglichkeit der Modularisierung entsteht bei der Verarbeitung von langen Operationen. Könnte eine Transaktion in mehrere Schritte aufgeteilt werden, so könnten mehrere Operationen einer Transaktion teilweise parallel ablaufen und so die Nebenläufigkeit entscheidend erhöht werden.

Wie man sieht, ist die Aufteilung von Transaktionen in mehrere Pakete im Hinblick auf die Kernproblemstellung dieser Arbeit durchaus von Vorteil. Das Konzept der *geschachtelten Transaktionen* bietet eine derartige Möglichkeit der Aufteilung und wird wie das der verteilten Transaktionen im Anschluss näher vorgestellt.

2.5.1 Verteilte Transaktionen

Verteilte Transaktionen sind im Prinzip ähnlich zu Transaktionen, die lokal ablaufen, nur mit dem Unterschied, dass sie aufgrund der über ein Netzwerk ablaufenden Kommunikation und die Autonomität der einzelnen Komponenten wesentlich komplexer handhabbar sind.

Im Normalfall ist es so, dass es bei der Ausführung einer (verteilten) Transaktion einen Koordinator und (mehrere) Teilnehmer (bzw. Operationen) gibt (siehe Abbildung 2.7). Teilnehmer selbst können wiederum Koordinatoren sein. Bezogen auf ein verteiltes System bedeutet dieser Hierarchieaufbau, dass es eine Applikation (Koordinator) gibt, die die verteilt auszuführenden Operationen (bei den Teilneh-

mern) im Kontext einer Transaktion regelt. Der Koordinator schickt beispielsweise RPC–Operationen (Remote Procedure Call-Operationen) an seine Teilnehmer und diese schicken die Resultate zurück.

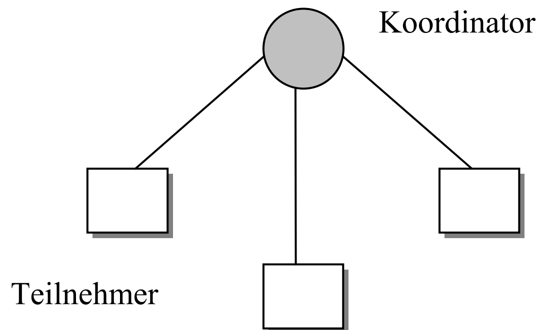


Abbildung 2.7: Verteilte Transaktion (nach [Litt04])

Nun liegt es aber in der Natur von Transaktionen, dass diese atomar ausgeführt werden – entweder alle Operationen werden ausgeführt, oder keine – , was einer genauen Koordination bedarf, um eine konsistente Terminierung der einzelnen Operationen zu gewährleisten. Ein derartiger Algorithmus, der gewährleistet, dass entweder Koordinator und Teilnehmer mit Commit, oder alle mit Abort terminieren, nennt man atomares Commitment Protokoll ACP (Atomic Commitment Protocol, nach [Bern87]). Geht man davon aus, dass jeder Teilnehmer entweder Ja (aus seiner Sicht gibt es keine Probleme) oder Nein (seinen Operation(en) kann/können nicht durchgeführt werden) wählen kann, ist ein ACP dafür verantwortlich, dass folgende Punkte eingehalten werden (nach [Bern87]):

1. Alle Prozesse müssen schlussendlich die gleiche Entscheidung treffen (Abort oder Commit)
2. Ein Prozess kann seine Entscheidung nicht ändern, sobald er eine gefällt hat (wenn ein Prozess „sagt“, er kann die Operation durchführen, dann kann er diese Entscheidung nicht mehr rückgängig machen.
3. Eine Commit-Entscheidung kann nur getroffen werden, wenn alle Prozesse Ja wählen
4. Wein keine Fehler auftreten und alle Prozesse wählen Ja, dann ist die Entscheidung Commit.
5. Passiert an irgendeinem Punkt der Ausführung ein Fehler (bspw. Systemabsturz), der den Algorithmus nicht verletzt und die Fehler wieder repariert, können alle Prozesse letzten Endes zu einer Entscheidung kommen.

Zusammenfassend bedeutet dies, dass wenn alle Teilnehmer (inkl. Koordinator) Ja wählen, die Transaktion committed wird. Wenn allerdings nur ein einziger Teilnehmer Nein wählt, wird die ganze Transaktion abgebrochen. Der letzte Punkt (5.) bedeutet, dass es möglich sein muss, schlussendlich zu einer Entscheidung zu gelangen, auch wenn während des Entscheidungsvorganges Fehler aufgetreten sind (Systemabstürze, Kommunikationsfehler, korrupte Nachrichten etc.). Die Erkennung solcher Fehler erfolgt im Normalfall über Timeouts.

Das einfachste und in den meisten Systemen eingesetzte ACP-Protokoll ist das Two-Phase Commit Protokoll. Two-Phase Commit deshalb, weil das Protokoll in zwei Phasen eingeteilt werden kann:

1. Voting-Phase
2. Entscheidungsphase

Wie in Abbildung 2.8 zu sehen ist, sendet der Koordinator in der Voting-Phase eine Voting-Anforderung an die einzelnen Teilnehmer (dies kann parallel erfolgen). Jeder Teilnehmer sendet anschließend seine Wahl (entweder Ja oder Nein) an den Koordinator zurück. In der Phase Zwei, der Entscheidungsphase, trifft der Koordinator dann seine Entscheidung aufgrund der gesammelten Stimmen. Sind diese alle Ja und die Wahl des Koordinators ist auch Ja, entscheidet er auf Commit und sendet dieses Ergebnis an alle Teilnehmer, die danach ihre Operationen ausführen. Sind nicht alle Stimmen Ja, wird Abort gesendet.

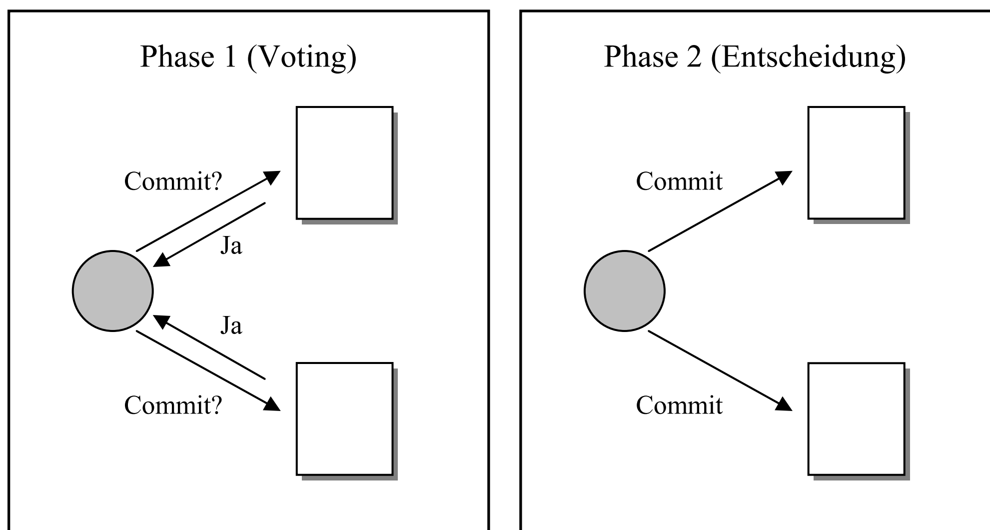


Abbildung 2.8: Phasen eines Two-Phase Commit (nach [Litt04])

Zwischen Phase Eins und Phase Zwei sind die einzelnen Teilnehmer in Ungewissheit. Außer natürlich jene Teilnehmer, die mit Nein gestimmt haben, da sie ja wis-

sen, dass ihr Nein zu einem Abbruch führt. Deshalb können diese auch sofort nach (oder vor) dem Senden ihrer Antwort abbrechen.

Die eben beschriebene Vorgangsweise des Two-Phase Commit ist bei lokaler, als auch bei verteilter Ausführung die gleiche, nur dass sich bei verteilten Anwendungen zusätzliche Probleme ergeben. Diese ergeben sich hauptsächlich daraus, dass Koordinator und Teilnehmer (teilweise) voneinander unabhängige Systeme sind und es deshalb vorkommen kann, dass der Koordinator oder ein/mehrere Teilnehmer während des Two-Phase Commit abstürzen, es zu Kommunikationsproblemen kommt oder ähnliches. Was also tun, wenn das passiert? (Anmerkung: Punkt 5 der ACP verlangt, dass es eine Entscheidung gibt).

Im Laufe des Two-Phase Commit gibt es mehrere Wartesituationen, auf die unterschiedlich reagiert werden kann:

- Teilnehmer wartet auf Vote-Anforderung:
  Wenn beispielsweise ein bestimmter Timeout erreicht wurde, kann der Teilnehmer jederzeit abbrechen, da er ja noch keine positive Zusage gemacht hat, an die er sich halten muss.
- Koordinator wartet auf Wahlergebnis:
  Hat er bis zum Timeout nicht alle Entscheidungen erhalten, kann der Koordinator keine positive Entscheidung treffen und sendet deshalb allen mit Ja geantworteten Teilnehmern einen Abort-Befehl.
- Teilnehmer hat gewählt und wartet auf Entscheidung:
  Dieser Fall ist der schwierigste, da der Teilnehmer (eine positive Wahl vorausgesetzt) nicht einfach alleine entscheiden kann. Er kann nicht einfach abbrechen, da es ja sein kann, dass der Koordinator allein ein Commit geschickt hat, aber vor dem Senden an diesen Teilnehmer abgestürzt ist, oder die Nachricht verloren gegangen ist. Genauso wenig kann er seine Operation(en) durchführen, da es ja durchaus möglich ist, dass ein anderer Teilnehmer Nein gewählt hat. Der Partizipant hat nur die Möglichkeit zu warten (zu blocken), bis er wieder eine Verbindung zum Koordinator herstellen kann, oder die Möglichkeit mit anderen Teilnehmern zu kommunizieren und eine Entscheidung zu treffen (Wenn beispielsweise ein anderer Teilnehmer Nein wählt, ist klar, dass die anderen ihre Operationen auch abbrechen können). Der zweite Ansatz wird auch kooperatives Terminierungsprotokoll genannt, setzt allerdings voraus, dass die Teilnehmer einander kennen. Die Kommunikation zwischen einzelnen Partizipanten ist jedoch aus Sicherheitsgründen oft nicht erwünscht.

Ein weiterer interessanter Aspekt ist das Verhalten eines Teilnehmers, wenn er während des Two-Phase Commit abstürzt. Er muss in diesem Fall dafür sorgen, dass er den letzten ihm bekannten Stand des Votingprozesses wiederherstellen kann. Welche Möglichkeiten es dazu genau gibt, wird in dieser Arbeit allerdings nicht besprochen und kann in Unterkapitel 7.4 von [Bern87] im Abschnitt „Recovery“ nachgelesen werden.

Weiters gibt es in der Praxis den Fall, dass ein Teilnehmer sein Commit-Versprechen nicht einhalten kann und es so zu einem inkonsistenten Zustand kommt. Der interessierte Leser wird auch hier auf [Litt04] verwiesen. (Stichwort „heuristic outcomes“).

Zur Optimierung des 2PC führen Öszu und Valduriez in [Öszu99] zwei Variationen des 2PC aus, nämlich das Presumed Abort 2PC Protokoll und das Presumed Commit 2PC Protokoll.

Beim Presumed Abort 2PC Protokoll wird davon ausgegangen, dass, wenn nach dem bereits erfolgten Voting-Prozess keine Informationen zur Transaktion im virtuellen Speicher des Koordinators vorhanden sind, von einem Abort der Transaktion ausgegangen wird. Die Annahme des Abbruchs bei keinem Vorliegen von Informationen kann getroffen werden, weil das Presumed Abort 2PC Protokoll vorsieht, dass bei einem Commit der Transaktion der Koordinator solange Informationen zur Transaktion behält, bis alle Teilnehmer den Commit bestätigt haben. Die Vorteile des Presumed Abort 2PC Protokolls liegen neben einer erwarteten höheren Effizienz darin, dass zum Einen der Koordinator die Transaktion bei einer Abort-Entscheidung sofort vergessen kann und zum Anderen der Nachrichtenaustausch zwischen Koordinator und Teilnehmer im Fall eines Abort verringert werden kann.

Das Presumed Commit 2PC Protokoll ist nahezu analog zum eben vorgestellten Presumed Abort 2PC Protokoll. Bei diesem Protokoll wird bei keinem Vorhandensein von Informationen über die Transaktion von einem Commit ausgegangen. Anders als beim Presumed Abort darf der Koordinator die Transaktion allerdings nicht sofort nach der Commit-Entscheidung vergessen, da ansonsten bei einem Absturz des Koordinators während des Votings Inkonsistenzen entstehen können (z.B. Entscheidung wäre Abort, aber keine Information über Transaktion vorhanden => Teilnehmer gehen von Commit aus). Stattdessen schreibt der Koordinator einen Commit-Eintrag in sein Log-File, sendet eine globale Commit Nachricht an alle Teilnehmer und darf erst danach die Transaktion „vergessen“. Im Falle eines Abort muss der Koordinator die Informationen der Transaktion so lange behalten, bis alle Teilnehmer den Abort bestätigt haben. Der Vorteil des Presumed Commit 2PC Protokolls liegt darin, dass

der Koordinator die Transaktion bei einem Commit nach Senden des globalen Commit sofort vergessen kann. Dieser Umstand bedeutet einerseits eine Reduktion der zu schreibenden Log-File Einträge (das Ende der Transaktion ergibt sich automatisch mit dem Eintrag der Commit-Entscheidung und muss nicht explizit erfasst werden) und andererseits muss der Commit von den Teilnehmern nicht mehr bestätigt werden.

Verteilte Transaktionen und ACID

Nachdem verteilte Transaktionen analog zu lokalen sind, können auch sie die ACID-Eigenschaften, sofern die von den Teilnehmern ausgeführten Operationen die ACID-Eigenschaften einhalten, erfüllen.

Atomizität wird durch ein ACP sichergestellt, Konsistenz ist weitgehend Sache der Applikation und die Erfüllung von Isolation und Dauerhaftigkeit betrifft größtenteils die Teilnehmer.

2.5.2 Geschachtelte Transaktionen

Das Konzept der geschachtelten Transaktionen bietet, wie zu Beginn dieses Unterkapitels kurz angedeutet, die Möglichkeit, Transaktionen in mehrere Teile aufzuspalten. Bei den geschachtelten Transaktionen wird zwischen geschlossenen und offenen unterschieden.

Eine geschachtelte Transaktion ist eine Top-Level Transaktion, innerhalb der atomare Transaktionen (so genannte Subtransaktionen) aufgerufen werden können. Diese Subtransaktionen können entweder flacher Natur sein, oder wiederum geschachtelt – sie können also genauso gut weitere Subtransaktionen aufrufen, sodass eine Baumstruktur entstehen kann. Die Tiefe der einzelnen Subtransaktionen ist nicht beschränkt und kann für jede Subtransaktion unterschiedlich sein. Veranschaulichen lässt sich eine derartige Verschachtelung am Besten anhand des Holiday Package Beispiels: Im Kontext einer Top-Level-Transaktion, die die gesamte Aktivität kontrolliert, werden die Subtransaktionen Flugbuchung, Hotelreservierung und Mietautoreservierung ausgeführt. Die Flugbuchung könnte dann beispielsweise wieder aus mehreren Subtransaktionen bestehen (so zum Beispiel die Buchung mehrere Flüge, falls Umstiege erforderlich sind oder bei der Reise zu mehreren Destinationen, oder einfach Hin- und Rückflug). Die Abbildung 2.9 soll den möglichen Aufbau visualisieren:

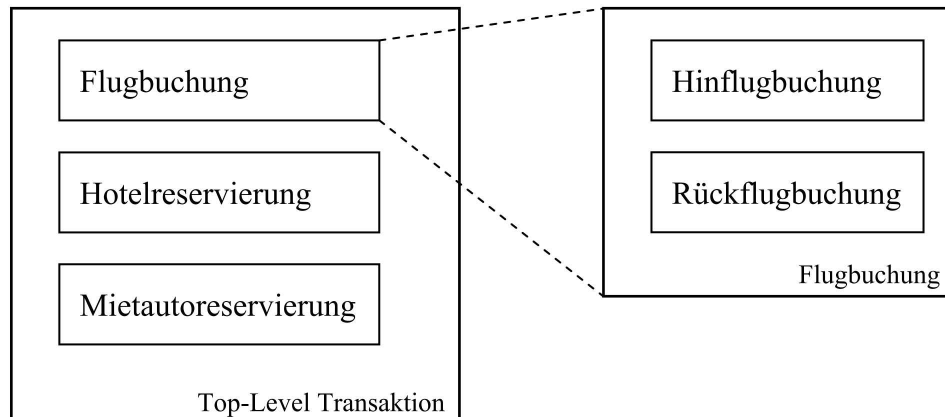


Abbildung 2.9: Beispiel einer geschachtelten Transaktion

Nachdem nun der grobe Aufbau von geschachtelten Transaktionen geklärt ist, können die wichtigsten Verhaltenseigenschaften dieser beleuchtet werden:

- Wird eine Subtransaktion mit Commit beendet, so sind ihre Resultate zunächst nur für den direkten Vorgänger (Parent) zugreifbar und sichtbar. Nach außen hin werden die Effekte erst aktiv, wenn alle direkten und indirekten Vorgänger auch mit Commit abschließen. Aufgrund der Induktion kann eine Subtransaktion also nur dann positiv abgeschlossen werden, wenn die Wurzeltransaktion commitet.
- Bei einem Abbruch einer (Sub-)Transaktion werden rekursiv alle Nachfolgetransaktionen abgebrochen, auch wenn diese bereits Commit signalisiert haben (nach außen hin waren sie ja noch nicht sichtbar).
- Wird eine Subtransaktion abgebrochen, so sind alle anderen Subtransaktionen außerhalb dieser vom Abbruch nicht betroffen. Für den Vorgänger der abgebrochenen Transaktion besteht nach einem Abbruch die Möglichkeit, entweder auch abzubrechen, oder die abgebrochene Transaktion erneut auszuführen.

Die Vorteile, die sich aus den Eigenschaften von geschachtelten Transaktionen ergeben, liegen auf der Hand. Durch das unabhängige Ausführen von Subtransaktionen können diese (teilweise) parallel ausgeführt werden. Weiters ist durch die Möglichkeit der Modularisierung eine bessere Anpassung der Transaktionen an heutzutage sehr populäre „Bausteinarchitekturen“ von Programmen möglich. Der größte Vorteil von geschachtelten Transaktionen liegt allerdings bei der Fehlerbehandlung. Während bei flachen Transaktionen alles verworfen werden muss, wenn ein Fehler auftritt, können bei geschachtelten Transaktionen bereits erfolgreich abgewickelte Teile

erhalten bleiben und die Durchführung der abgebrochenen Teile kann erneut versucht werden. Dies ist insbesondere bei langen Transaktionen von immenser Wichtigkeit, da es bei diesen umso mühsamer wäre, diese vom Status „Null“ wieder zu beginnen.

So viele Vorteile mit der Verwendung von geschachtelten Transaktionen einhergehen, gibt es besonders im Hinblick auf die Kernproblemstellung dieser Arbeit noch einige weitere Punkte, die eine optimale Lösung des Problems gewährleisten könnten. Eine solche Erweiterung des bisher vorgestellten Konzepts der geschachtelten Transaktionen stellt das Konzept der offenen geschachtelten Transaktionen dar, das in weiterer Folge vorgestellt wird. (Anmerkung: das bisher besprochene Konzept wird auch geschlossene geschachtelte Transaktionen genannt)

Eine weitere mögliche Variante bei den geschachtelten Transaktionen ist die Einteilung der Subtransaktionen in vitale und nicht vitale. Während der Abbruch einer vitalen Subtransaktion wie üblich zu einem Abbruch der gesamten Transaktion führt, ist der Abbruch einer nicht vitalen Transaktion für das Resultat der Transaktion nicht relevant. Diese Unterscheidung ist bei der Modellierung von Geschäftsprozessen sehr nützlich. Beispielsweise ist es bei der Urlaubsbuchung unerlässlich, ein Hotelzimmer und einen Flug zu bekommen (diese beiden sind vital), wohingegen die Nichtdurchführbarkeit einer Mietauto-Buchung nicht so tragisch ist (nicht vital) und daher die Urlaubsbuchung trotzdem durchgeführt werden kann.

Offene geschachtelte Transaktionen

Bei geschlossenen geschachtelten Transaktionen werden die Resultate von Subtransaktionen erst nach außen hin sichtbar, wenn die ganze Transaktion mit Commit abgeschlossen wird. Dies ist aber in einigen Fällen nicht wünschenswert, da dieser Umstand die Nebenläufigkeit stark einschränkt beziehungsweise die Nebenläufigkeit durch eine Sichtbarmachung entscheidend erhöht werden könnte, was vor allem bei stark frequentierten Web-Applikationen notwendig ist. Zurückkommend auf das Holiday Package ist es zwar ein erheblicher Vorteil, mit geschachtelten Transaktionen eine Aufteilung vornehmen zu können, doch da die Änderungen erst am Ende abgeschlossen und sichtbar werden, müssen alle Subtransaktionen solange innehalten, bis die „langsamste“ Subtransaktion abgeschlossen ist. Ist also die Flugbuchung (intern) bereits abgeschlossen und die Hotelreservierung noch im Gange, könnte die Subtransaktion alle blockierten Datensätze und Ressourcen freigeben und die Reservierung für andere Transaktionen sichtbar gemacht werden.

Genau diese Erweiterung wird bei offenen geschachtelten Transaktionen vollzogen. Sie bieten die Möglichkeit, die Ergebnisse von Subtransaktionen für andere „offen zu legen“, die Subtransaktionen also vorzeitig zu commiten (pre-commit). Dabei kann für jede einzelne Subtransaktion festgelegt werden, ob sie nach außen hin sichtbar, oder nach außen hin nicht sichtbar sein soll.

Die verfrühte Sichtbarmachung von Subtransaktionen stellt bei der positiven Beendigung der gesamten Transaktion kein Problem dar, doch würde diese im Fall eines Abbruchs der Top-Level Transaktion zu einer Verletzung der Atomizität führen, da ja einige Teile der Transaktion persistent gemacht würden beziehungsweise Auswirkungen auf persistente Daten hätten. Die Atomizität verlangt allerdings, dass eine Transaktion entweder vollkommen oder überhaupt nicht durchgeführt wird.

Um diese Eigenschaft dennoch zu erfüllen, müssen die Resultate wieder rückgängig gemacht werden. Dies kann allerdings nicht einfach durch herstellen des vorherigen Zustandes passieren, da es möglich und denkbar ist, dass andere Transaktionen die neuen, von den pre-commiteten Subtransaktionen hervorgerufenen Resultate bereits wieder verändert haben. Es ist daher notwendig, eine Transaktion S^{-1} auszuführen, die invers zur pre-commiteten Subtransaktion S ist. Die Transaktion S^{-1} wird *kompensierende Transaktion* genannt. Wurden beispielsweise 100 € von Konto A abgebucht und auf Konto B gutgeschrieben, muss die kompensierende Transaktion genau das Umgekehrte machen, nämlich 100 € auf das Konto A gutschreiben und 100 € von Konto B abbuchen. Kompensierende Transaktionen müssen aber nicht flach sein, sondern können ihrerseits auch geschachtelt sein.

Durch kompensierende Transaktionen muss sichergestellt werden, dass die aufgetretenen Effekte völlig beseitigt werden, so als hätte die gesamte Transaktion niemals stattgefunden. Wird dies erreicht, ist die Atomizität gewahrt.

Geschachtelte Transaktionen und ACID

- Atomicity: Wie soeben erwähnt, kann die Atomizität bei offenen geschachtelten Transaktionen durch kompensierende Transaktionen erreicht werden. Geschlossene geschachtelte Transaktionen erfüllen ebenfalls das Prinzip der Atomizität, da die einzelnen Subtransaktionen atomar sind, die Top-Level-Transaktion atomar ist und bei einem Abort alle Subtransaktionen abgebrochen werden. Bei der Verwendung von nicht vitalen Subtransaktionen wird das Prinzip der Atomizität hingegen gelockert, da bei erfolgreicher Beendigung der gesamten Transaktion und einem Abbruch einer nicht vitalen Subtransaktion der Ausgang nicht atomar ist.

- Consistency: Nachdem es Aufgabe der Applikation ist, die Konsistenz sicherzustellen trifft auch diese Eigenschaft auf die beiden Konzepte zu.
- Isolation: Durch die Sichtbarmachung von (Teil-)Ergebnissen wird das Prinzip der Isolation bei offenen geschachtelten Transaktionen gelockert und somit im Hinblick auf die Definition der Eigenschaft nicht erfüllt. Jedoch existieren etliche Protokolle, um die Ausführung von geschachtelten Transaktionen zu synchronisieren [Elma92].
- Durability: Streng genommen wird die Dauerhaftigkeitsbedingung von geschachtelten Transaktionen nicht erfüllt, da eine commitete Subtransaktion einerseits solange keine Effekte hat, solange nicht die Vorgängertransaktion auch commitet und andererseits, weil ein Rollback einer sich irgendwo in der Hierarchie befindlichen Transaktion zu einem Rollback all ihrer Subtransaktionen führt (nach [Gray93]). Weiters wird bei offenen geschachtelten Transaktionen die Dauerhaftigkeitsbedingung in der Weise gelockert, dass diese Transaktionen bei einem Rollback ihrer Vorgänger trotzdem persistent bleiben.

Kapitel 3

Transaktionsprotokolle und Frameworks für Web Services

In diesem Kapitel werden Transaktionsprotokolle und Frameworks für Web Services vorgestellt. Diese sind einerseits speziell für Web Services konzipiert oder werden andererseits hinsichtlich ihrer Eignung für Web Services betrachtet.

Um der uninformierten LeserIn zu ermöglichen, die Inhalte zu verstehen, wird nun das Konzept von Web Services und deren wichtigste Komponenten in Kurzform erläutert. Die Inhalte stammen dabei im Wesentlichen aus [Rahm03] und können dort genauer nachgelesen werden.

3.1 Web Services und Transaktionsmanagement allgemein

Ein Web Service ist im Wesentlichen ein Dienst, der von einem Anbieter einem oder mehreren Clients zur Verfügung gestellt wird und der darauf ausgerichtet ist, von Maschinen vollautomatisch verarbeitet werden zu können. Beispiele für derartige Dienste sind der bestehende Übersetzungsdienst „BabelFish" von AltaVista oder einfach ein Flugbuchungsdienst, der von einer Airline zur Verfügung gestellt wird.

Ein wichtiger Anspruch von Web Services ist es, plattform- und programmiersprachenunabhängig zu sein. Diese Interoperabilität wird durch XML-basierte Standards erreicht. Daten und Informationen werden im XML-Format über HTTP ausgetauscht (es können beispielsweise aber auch FTP oder SMTP verwendet werden). Auch die

Schnittstellenbeschreibung eines Dienstes erfolgt via XML. Die wichtigsten im Zusammenhang mit Web Services hier vorgestellten Standards sind SOAP (Simple Object Access Protocol), WSDL (Web Service Definition Language) und UDDI (Universal Description, Discovery and Integration).

SOAP ist das Protokoll, mittels dem der Client und der Diensteanbieter kommunizieren, WSDL wird zur Beschreibung eines Web Services verwendet (z.B. wie es erreichbar ist, benötigte Parameter, Returnwerte usw.) und UDDI stellt einen Mechanismus dar, um Web Services publik zu machen und bietet für den Client die Möglichkeit, gezielt nach Web Services zu suchen und genaue Informationen über die gefundenen Services zu erhalten (WSDL). Die Kommunikation erfolgt auch hier via SOAP. Wie die Interaktionen und Verwendungen der Protokolle aussehen können, ist in Abbildung 3.1 (aus [Rahm03]) zu sehen.

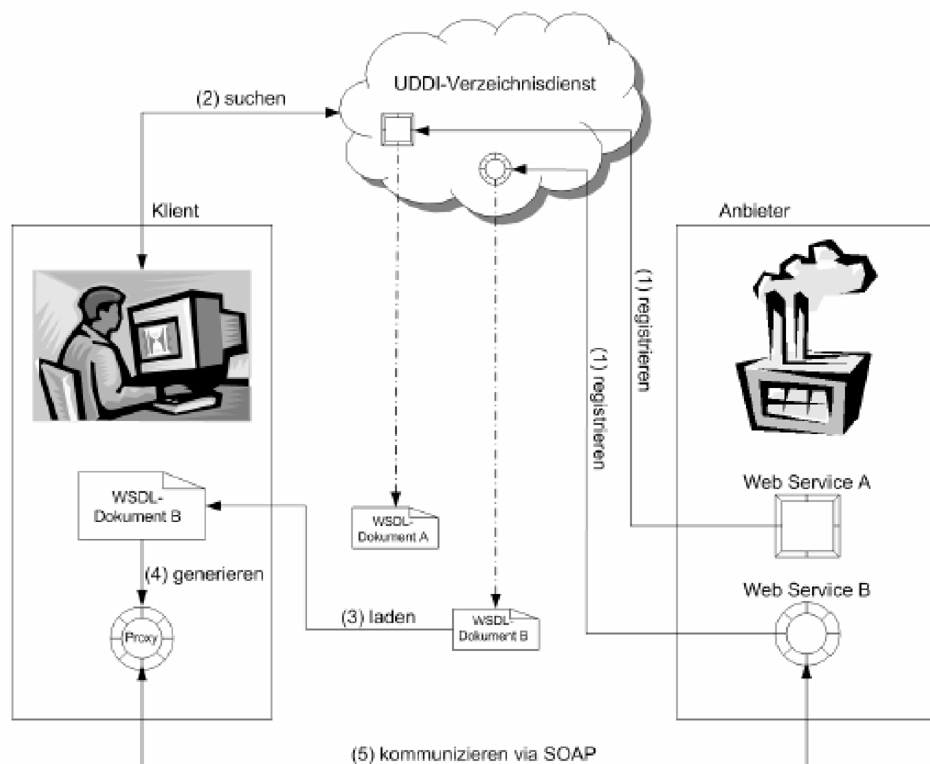


Abbildung 3.1: Überblick über den Einsatz von Web Services (aus [Rahm03])

3.1.1 SOAP

SOAP ist ein Kommunikationsprotokoll, das sowohl für die einfache Zustellung von Nachrichten, als auch für RPC-Aufrufe verwendet werden kann. Der grundsätzliche Aufbau einer SOAP-Nachricht ist in Abbildung 3.2 dargestellt.

```
<soap:Envelope   soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header>
   <!--Der Header ist optional -->
  </soap:Header>

<soap:Body>
   <!-- Serialisierte Objektdaten -->
  </soap:Body>
</soap:Envelope>
```

Abbildung 3.2: Aufbau einer SOAP-Nachricht (aus [Rahm03])

Innerhalb des Wurzelelements `<envelope>` befindet sich das `<body>` Element. Im `<body>` Element befinden sich typischerweise serialisierte Daten. Wie die Abbildung von Objekten auf XML erfolgt, wird durch das encodingStyle-Attribut (siehe Abbildung 3.2) angegeben.

Die Standard-Serialisierung von SOAP bietet die gebräuchlichsten Datentypen (Integer, String,…) und weiters auch die Möglichkeit zusammengesetzter Datentypen (z.B. Adressen). Auch die Darstellung von Arrays wird durch SOAP definiert. Reichen diese Standardtypen nicht aus, müssen entsprechende neue Typen festgelegt werden.

Der optionale Header einer SOAP-Nachricht dient zur individuellen Erweiterung von SOAP, wenngleich SOAP bereits einige Erweiterungsattribute für Authentifizierung, Transaktionsmanagement etc. definiert. Im Header können also Informationen, die beispielsweise für die Verarbeitung der Nachricht relevant sind, abgelegt werden (Signaturen, Transaktionsinformationen, etc). Mehr dazu bei den Transaktionsprotokollen.

3.1.2 UDDI

Bevor ein Web Service genutzt werden kann, muss es dem Client bekannt gemacht werden. Dies kann entweder auf direktem Weg passieren (Bekanntmachung auf einer Website etc.) oder wie in Abbildung 3.1 über ein UDDI-Verzeichnis, bei dem das Web Service registriert werden kann. Der Client kann dann in einem UDDI-Verzeichnis nach Diensten suchen (z.B. nach einem Übersetzungsdienst). Die Kommunikation zwischen Client und Verzeichnis erfolgt dabei mittels SOAP. Wird ein entsprechender Dienst gefunden, können die technischen Informationen, die als Referenz beim UDDI-Verzeichnis aufliegen – aber irgendwo im Web gespeichert sein können – abgerufen werden. Mittels der in einem WSDL-Dokument enthaltenen

Informationen ist es dem Client dann möglich, geeignete SOAP-Nachrichten zu generieren oder – wie in Abbildung 3.1- von einem Proxy generieren zu lassen und diese dann an den Diensteanbieter zu senden (beispielsweise via HTTP-POST Anfrage), der dieses File dann verarbeiten kann und entsprechende Handlungen setzt (etwa einen übersetzten Text via SOAP wieder zurücksendet oder einen Flug bucht).

3.1.3 WSDL

In einem WSDL-Dokument gibt es sechs verschiedene Elementarten, die zur Beschreibung eines Dienstes verwendet werden: types, messages, portTypes, bindings, ports und services.

Mittels dieser Elemente werden Datentypen, Nachrichten (und deren Bestandteile), Operationen (bestehend aus einer oder mehreren Nachrichten) und Bindungen der abstrakten Operationen an konkrete Nachrichtenformate, Protokolle (z.B. SOAP) und Netzwerkprotokolle (z.B. HTTP) definiert. Die Bindungen werden mit konkreten Netzwerkadressen zu Ports zusammengefasst, welche wiederum die Bestandteile eines Services darstellen.

Bei den Operationen werden vier Primitiven unterschieden (nach [Rahm03]):

- One Way: nur Endpunkt empfängt Nachricht
- Notification: nur Endpunkt sendet eine Nachricht
- Request-Response: Endpunkt empfängt zuerst eine Nachricht und sendet dann eine korrellierte Nachricht als Antwort
- Solicity-Response: Endpunkt sendet zuerst eine Nachricht und empfängt dann eine korrellierte Antwort

Natürlich ist es auch möglich, in einem Web Service auf andere Web Services zurückzugreifen (beispielsweise verwendet ein Dienst einen Deutsch-Englisch und einen Englisch-Spanisch Übersetzungsdienst, um Wörter vom Deutschen ins Spanische zu übersetzen). Dies nennt man Komposition und kann im Detail in [Rahm03] nachgelesen werden.

3.1.4 Transaktionsmanagement bei Web Services allgemein

Möchte man ein oder mehrere Web Service(s) innerhalb einer Transaktion ausführen, muss dies erstens dem/den Web Service/Web Services mitgeteilt werden und zweitens muss der Ausgang der Transaktion (und das damit verbundene Verhalten der beteiligten Web Services) unter der Initiatorapplikation und den teilnehmenden

Services abgestimmt werden. Diese Abstimmung erfolgt im Allgemeinen – wie auch bei den im Anschluss vorgestellten Transaktionsprotokollen – über einen Koordinator, der als eigenes Service laufen kann, oder in der Initiatorapplikation integriert sein kann und der für die Durchführung der Transaktion verantwortlich ist.

Um einem Web Service mitzuteilen, dass es an einer Transaktion teilnehmen soll, wird diesem ein so genannter Context, der Informationen über die Transaktion enthält, via XML im SOAP Header des Web Service Aufrufs übermittelt. Nach Erhalt der Context-Information weiß der Web Service Anbieter, dass das Web Service innerhalb der im Context angegebenen Transaktion ausgeführt werden soll und registriert dazu beim Koordinator (Informationen über diesen sind im Context enthalten) einen Partizipant. Die Abwicklung der Transaktionslogik – z.B. Two-Phase Commit – erfolgt sodann zwischen Koordinator und Teilnehmern, die aufgrund der ausgetauschten Nachrichten die nötigen Handlungen das Web Service betreffend – wie etwa Durchführung einer Hotelbuchung via lokaler Datenbanktransaktion – setzen beziehungsweise an die Applikation des Teilnehmers weiterleiten muss.

Das genaue Aussehen der Context-Informationen beziehungsweise des Transaktionsnachrichtenflusses ist abhängig vom verwendeten Transaktionsprotokoll. Die Kommunikation zwischen Koordinator und Teilnehmer(n) erfolgt im Normalfall via SOAP.

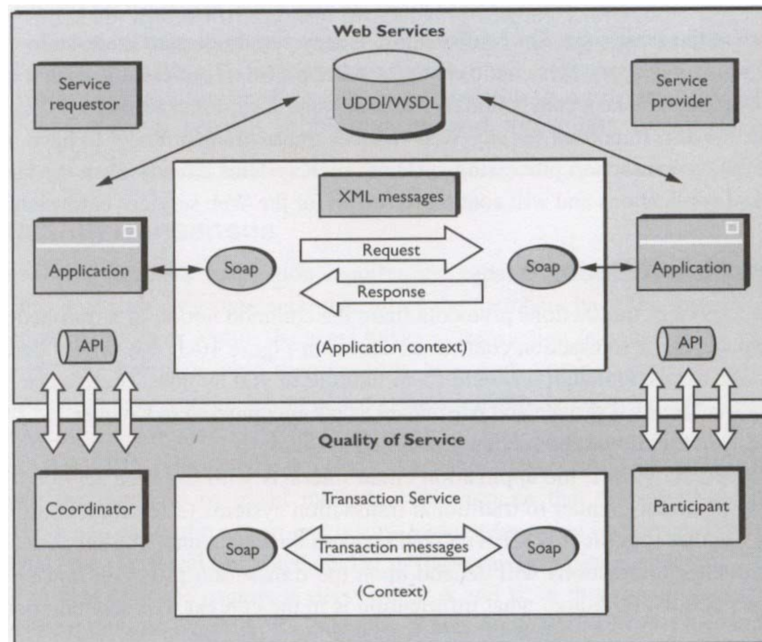


Abbildung 3.3: Web Services, Transaktionen und Context (aus [Litt04])

Nachdem nun das grobe Konzept des Transaktionsmanagements mit Web Services - in Abbildung 3.3 (aus [Litt04]) grafisch dargestellt - bekannt ist, können wir uns nun den einzelnen Transaktionsprotokollen zuwenden. Diese sind OASIS Business Transaction Protocol (BTP), Web Services Coordination/Web Services Transaction (WS-C/WS-Tx) sowie das Web Services Composite Application Framework (WS-CAF).

3.2 OASIS Business Transaction Protocol

Das erste für Transaktionen mit Web Services geeignete Protokoll, das in dieser Arbeit behandelt wird, ist das OASIS Business Transaction Protocol, an dem im Jahr 2001 von einem Konsortium (unter anderem mit den Mitgliedern HP, Oracle und BEA) zu arbeiten begonnen wurde und die erste Spezifikation im April 2004 fertig gestellt wurde [Litt04].

Der letzte und für diese Arbeit herangezogene Spezifikations-Working Draft [Furn04] in der Version 1.1 stammt vom November 2004. Neben dieser Referenz wurden für dieses Unterkapitel auch noch [Litt03], [Litt04] und [Pott02] herangezogen. Da die Spezifikation an die 200 Seiten hat, kann leider nur auszugsweise auf die Inhalte eingegangen werden.

Wie bereits in der Theorie bei den geschachtelten Transaktionen angesprochen, ist es schwierig, eine Business Transaktion wie das Holiday Package Beispiel - das ja einen Geschäftsprozess darstellt – mit traditionellen Transaktionskonzepten durchzuführen. Es ist bei derartig langen Transaktionen daher nötig, die ACID-Bedingungen auf die eine oder andere Art zu lockern, um einzelne Datensätze, Tabellen, Datenbanken oder Applikationen nicht zu blockieren.

Um dies zu erreichen, setzt das BTP auf den so genannten Two-Phase Outcome. Dieser Two-Phase Outcome (2PO) ist ähnlich dem bereites kennen gelernten Two-Phase Commit (2PC), der üblicherweise bei verteilten Transaktionen angewendet wird. Während der 2PC grob aus den Operationen Begin (oder Prepare) und Commit bzw. Rollback besteht (siehe Abschnitt 2.5.1), heißen diese Operationen beim 2PO Prepare und Confirm bzw. Cancel. Der große Unterschied zwischen 2PC und 2PO besteht nun darin, dass beim 2PC alle Teilnehmer blocken müssen, bis der Koordinator eine Entscheidung trifft und die Zeit zwischen dem Beginn der Transaktion und der Entscheidung (Commit oder Rollback) versucht wird möglichst kurz zu halten, während diese beiden Punkte für den 2PO nicht zutreffen.

Sendet der Koordinator beim 2PO Prepare an die Teilnehmer, steht es jedem Teilnehmer frei, wie sein Verhalten auf diese Aufforderung ist. Er muss zwar signalisieren, dass er „Prepared“ hat, doch kann er die Operation bereits ganz durchführen und beim späteren Erhalt von Cancel (Transaktion abbrechen) diese Inkonsistenzen (contradictions genannt) durch eine entsprechende kompensierende Transaktion wieder beheben oder ganz traditionell blocken. Welche Möglichkeiten jeder einzelne Teilnehmer beim Erhalt einer Prepare-Aufforderung hat – für welche provisorischen Effekte er sich entscheidet – ist zusammen mit den daraus resultierenden Handlungen beim anschießenden Erhalt von Confirm oder Cancel in Tabelle 3.1 (nach [Furn04]) aufgelistet.

Provisorischer Effekt	**Endgültiger Effekt (bei Confirm)**	**Gegeneffekt (bei Cancel)**
Speichern der beabsichtigten Änderungen, ohne sie durchzuführen	Änderungen durchführen	Gespeicherte Änderungen löschen
Änderungen ausführen und sichtbar machen; Informationen zum Rückgängig machen speichern	Informationen zum Rückgängig machen löschen	Annulierungsaktion durchführen (etwa kompensierende Operation)
Originalstatus speichern, Zugriffe von außen sperren(locking), Änderungen durchführen (Anm.: typisches Verhalten wie bei Datenbanken)	Zugriff wieder erlauben	Originalstatus wiederherstellen
Änderungen durchführen, als provisorisch markieren, sichtbar machen	Als endgültig markieren	Löschen oder als abgebrochen markieren

Tabelle 3.1: Einige Alternativen für provisorische, endgültige und Gegeneffekte

Der große Vorteil der freien Entscheidungsmöglichkeit gegenüber dem 2PC ist der, dass dadurch von Teilnehmer zu Teilnehmer unterschiedliche, jeweils angemessene Arten der Implementierungen möglich sind und beispielsweise durch das mögliche

vorzeitige Publikmachen von Effekten die Nebenläufigkeit entscheidend erhöht werden kann.

Das Problem bei der freien Entscheidungsmöglichkeit der Teilnehmer ist allerdings, dass von Seite des Koordinators keine Aussage über die Korrektheit einer Transaktion und deren genauem Verhalten getroffen werden kann (Stichwort ACID; siehe weiter unten).

Da die Applikation des Koordinators frei entscheiden kann, wie viel Zeit zwischen der ersten und der zweiten Phase (zwischen Prepare und Confirm bzw. Cancel) liegt, wird das 2PO-Protokoll auch als *open top completion protocol* bezeichnet (2PC ist ein *closed top completion protocol*).

Wie bereits in der allgemeinen Einführung in Web Services vorgestellt, gibt es neben den Applikationen, die für die Geschäftslogik und Ausführung von Web Services zuständig noch Koordinatoren und Teilnehmer, die jeweils mit den Applikationen verknüpft sind und die das Transaktionsservice bilden.

Auch beim BTP gibt es diese Trennung, wenngleich diese etwas unsauber ist, wie wir später noch feststellen werden. Abbildung 3.4 zeigt eine einfache Koordinator:Teilnehmer-Beziehung (Superior:Inferior-Beziehung).

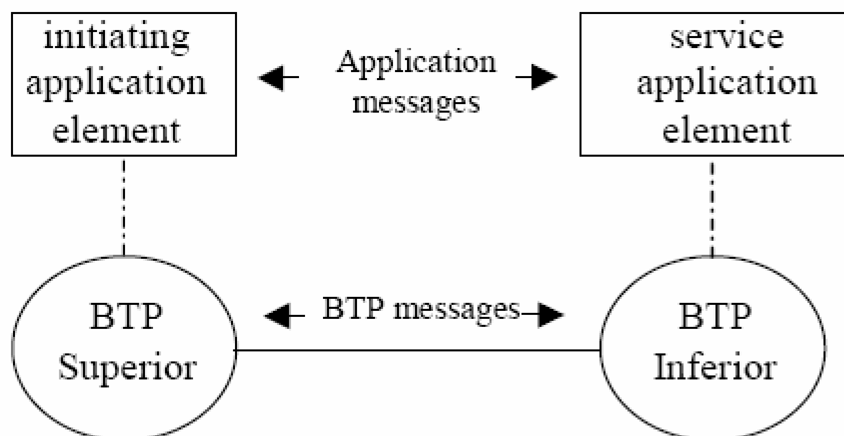


Abbildung 3.4: Superior:Inferior Beziehung (aus [Furn04])

Der Superior kann natürlich mehrere Inferior haben. Selbstverständlich ist – wie bei den verteilten Transaktionen schon besprochen – auch eine Hierarchiebildung möglich. Tritt ein Inferior wiederum als Superior auf, so nennt man dies Interposition. Abbildung 3.5 (nach [Furn04]) zeigt einen derartigen Transaktionsaufbau.

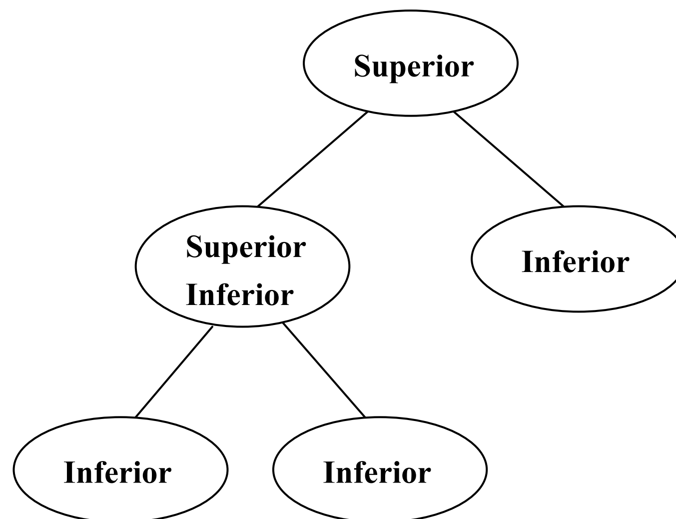


Abbildung 3.5: Transaktion mit einem Intermediate (interposition) - (nach [Furn04])

Erwähnenswert im Zusammenhang mit Interposition ist die Tatsache, dass Interposition und Schachtelungen (nesting) nicht das gleiche sind. Anders als bei geschachtelten Transaktionen liegt bei Interposition nur eine einzige Transaktion vor. Kann eine Operation nicht ausgeführt werden, ist davon die ganze Transaktion betroffen. Bei geschachtelten Transaktionen hingegen ist dies nur die jeweilige Subtransaktion, wodurch bereits erledigte Teile (andere Subtransaktionen) erhalten bleiben können und lediglich diese Subtransaktion wiederholt oder abgebrochen werden muss, ohne wie bei der Interposition alle anderen erfolgreichen Operationen zu verlieren.

Wie die Kommunikation zwischen Applikation und dem jeweiligen BTP-Element abläuft, wird von BTP nicht spezifiziert und steht somit jedem frei. Weiters muss jeder Akteur (jeder Service Anbieter und der Koordinator) das jeweilige BTP-Element zur Verfügung stellen, sprich wenn ein Service an einer Transaktion nach BTP teilnehmen können soll, muss der Service Anbieter für eine entsprechende Inferior-Schnittstelle sorgen. Der Austausch von BTP Nachrichten erfolgt über XML. Welche Inhalte diese Nachrichten haben müssen, wird durch XML-Schemata spezifiziert. Welche Nachrichten es genau gibt und wie diese im Detail aussehen, würde hier zu weit gehen. Die interessierte LeserIn wird daher auf die entsprechenden Kapitel in der Spezifikation [Furn04] verwiesen.

Für die Übertragung der XML-Nachrichten gibt es ein entsprechendes HTTP-SOAP1.1-Binding. Enthalten in dieser SOAP/HTTP-Binding Spezifikation sind Regeln, die es einer Applikation erlauben, den Beginn einer Transaktion (durch die Übermittlung eines neuen CONTEXT) oder die Registrierung eines Inferiors (genannt ENROL), die im SOAP-Header transportiert werden, mit einer Applikations-

nachricht (z.B. Serviceaufruf) zu assoziieren oder zu verbinden, die im SOAP-Body transportiert wird (sinngemäß nach [Furn04]). In Listing 3.1 ist dies in abstrakter Form beispielhaft dargestellt.

Listing 3.1:

```
<soap>
     <header>
          <!-- neuer Context(neue Transaktion) -->
     </header>
     <body>
          <!-- Service Aufruf (z.B. Flug buchen) -->
     </body>
</soap>
```

Die Bindung für SOAP via HTTP ist auf den ersten Blick zwar ausgezeichnet, da diese genau ins Konzept der Web Services passt, doch stellt die BTP-Spezifikation den Anspruch, nicht ausschließlich ein Protokoll für Web Services zu sein. Deshalb ist die SOAP/HTTP Bindung in der Spezifikation explizit nur optional. Dies eröffnet zwar die Möglichkeit, sich das Übertragungsprotokoll aussuchen zu können, bedeutet allerdings auch, dass es kein standardisiertes Protokoll gibt und die für die Interaktion von heterogenen Systemen so wichtige Einheitlichkeit damit schwer zu erreichen ist beziehungsweise verloren geht. Dieser Umstand könnte ein entscheidender Nachteil für das Business Transaction Protocol sein, sich anderen Protokollen gegenüber durchzusetzen und Standard zu werden.

3.2.1 Transaktionsunterstützung

Wie bereits in der Transaktionstheorie festgestellt, bedeutet Atomizität, dass entweder alle Operationen einer Transaktion ausgeführt werden oder keine. Hält man sich nun allerdings das Holiday Package Beispiel vor Augen, so kann es sein, dass beispielsweise ein passender Flug gefunden, dieser gebucht, aber später ein günstigerer Flug gefunden wird und es somit nicht sinnvoll wäre, beide Flüge zu buchen, sondern nur einen der beiden. Um den Anforderungen - wie einer gelockerten Atomizität - von Geschäftsprozessen zu genügen, gibt es im BTP zwei verschiedene Transaktionsarten: Atoms und Cohesions. Beide verwenden das zuvor vorgestellte open top completion protocol.

Atoms: Atoms sind ähnlich den klassischen Transaktionsformen. Das Ergebnis aller Beteiligten muss konsistent sein. Entweder alle bestätigen und kommen zu einem positiven Ergebnis (Confirm) oder alle Beteiligten brechen ab (Cancel). Kann nur ein einzelner Teilnehmer nicht bestätigen, wird die ganze Transaktion abgebrochen.

Cohesions: Bei dieser Transaktionsform ist ein konsistentes Ergebnis nicht notwendig. Der Koordinator (bei Cohesions „Cohesion Composer" genannt) kann während der Transaktion festlegen, welche Operationen mit Confirm und welche mit Cancel beendet werden. Die Menge der zum Confirm festgelegten Operationen nennt man confirm-set. Das confirm-set wird, sobald es festgelegt wurde wie atoms behandelt – es muss also ein atomarer Ausgang unter den Operationen erzielt werden.

Durch die Verwendung des open top completion protocols ist es bei beiden Transaktionsarten nicht möglich, alle ACID Prinzipien zu erfüllen. Ob und in welcher Form die einzelnen Eigenschaften erfüllt werden, ist in Tabelle 3.2 (nach [Pott02]) aufgelistet. Zum Vergleich sind in der Tabelle auch die Eigenschaften von klassischen ACID-Transaktionen angeführt.

Eigenschaft	**Klassische ACID-Transaktion**	**BTP Atoms**	**BTP Cohesions**
Atomicity	Alles oder nichts	Alles oder nichts	Vereinbart zwischen Teilnehmer und Koordinator[4]
Consistency	Saubere Statuswechsel	Saubere Statuswechsel	Sobald das confirm-set festgelegt wurde, ist die Konsistenz gewährleistet
Isolation	Effekte sind bis zur Übereinkunft aller nicht sichtbar	gelockert; Effekte können sichtbar sein, Service entscheidet darüber	gelockert; Effekte können sichtbar sein, Service entscheidet darüber
Durability	Effekte dauerhaft	Effekte dauerhaft	Effekte dauerhaft, einige können vorübergehend sein

Tabelle 3.2: BTP und ACID (nach [Pott02])

[4] wenn das confirm-set alle Teilnehmer umfasst, sind auch cohesions atomar

Wie die Tabelle 3.2 zeigt, kann bei BTP Atoms lediglich die Isolation nicht garantiert werden. Dadurch kann es zu vorübergehenden Inkonsistenzen kommen, so etwa wenn das Service die Operation bereits durchführt, aber anschließend eine Cancel-Nachricht vom Koordinator gesandt wird oder es bei der Durchführung (wenn Confirm erhalten wurde) zu Problemen kommt. Dieser Widerspruch erfordert eine Publikmachung beim Koordinator um ihn wieder beheben zu können. Die genaue Vorgehensweise dafür ist in der Spezifikation [Furn04] geregelt (Stichworte: Contradiction und Hazard).

Bei den BTP Cohesions sind nach der Festlegung des confirm-set die Eigenschaften Konsistenz und Dauerhaftigkeit erfüllt, wobei es zur vorübergehenden Dauerhaftigkeit von Effekten kommen kann, die schlussendlich nicht im confirm-set enthalten sind.

Qualifier

Neben den bisher genannten Transaktionsmechanismen gibt es zusätzlich die so genannten Qualifier. Diese stellen eine Möglichkeit dar, weitere Transaktionsinformationen zu übertragen. So ist es einem Teilnehmern etwa möglich, über einen Qualifier anzugeben, für wie lange er seinen Status beibehält oder wie lange er bereit ist, an der Transaktion teilzunehmen. Das große Problem bei den Qualifiern, die ja grundsätzlich eine gute Idee sind, ist aus Implementierungssicht, dass diese Informationen in der Transaktionslogik verankert sind, obwohl sie eigentlich für die Geschäftslogik relevant sind. Soll heißen: Der Koordinator der Transaktion erhält Informationen, die eigentlich von der Applikation benötigt werden. Details zu Qualifiern können in [Furn04] nachgelesen werden.

Optimierungen

Gibt es nur einen Transaktionsteilnehmer, so ist es nicht nötig, ein Zwei-Phasen-Protokoll zur Abstimmung der Teilnehmer zu verwenden. Zu diesem Zweck gibt es die Möglichkeit der „One phase confirmation“, bei der dem (einzigen/einzig verbliebenen) Teilnehmer lediglich die Nachricht CONFIRM_ONE_PHASE gesendet wird (Im Gegensatz zum Erhalt von PREPARE, senden von PREPARED und anschließendem Erhalt von CONFIRM). Für Details zu dieser und weiterer Optimierungen wie „Spontaneous repeat“, „One Shot“, „Resignation“, „Autonomous cancel, autonomous confirm and contradiction“ wird die interessierte LeserIn wieder auf [Furn04] verwiesen.

3.2.2 BTP und Holiday Package

Nachdem das Konzept und die Möglichkeiten von BTP vorgestellt wurden, kann nun die Eignung des Protokolls hinsichtlich des Holiday Package Beispiels untersucht werden.

Von der Seite der Transaktionsmöglichkeiten und der Möglichkeit der Interposition gesehen eignet sich BTP eigentlich recht gut für das Beispiel. Durch die Cohesions ist es möglich, flexibel auf Interaktionen zu reagieren und so das Suchen des optimalen Packages zu gewährleisten und schlussendlich – wenn es gefunden wurde – dieses confirm-set atomar auszuführen. Weiters ist durch den Two-Phase Outcome erstens eine Abstimmungsmöglichkeit unter den einzelnen Teilnehmern gegeben und zweitens kann durch die Möglichkeit, die das 2PO Protokoll bietet – das individuelle Entscheiden der einzelnen Services (Serviceanbieter) über die Art und Weise der Verarbeitung von Serviceaufrufen – eine gezielte Optimierung auf Ebene der Services etwa durch die Steigerung der Nebenläufigkeit durch vorzeitiges Sichtbarmachen vorläufiger Effekte stattfinden. Dass derartige Optimierungen bei großen Zugriffszahlen, wie sie bei vielen Buchungen auftreten können, notwendig sind, steht außer Zweifel.

Neben den bisher genannten positiven Aspekten von BTP hinsichtlich des Holiday Packages erweisen sich auch Qualifier für die Modellierung eines derartigen Geschäftsprozesses als nützlich. Im Zusammenhang mit dem open top completion protocol ist es von Serviceseite her möglich, Reservierungen für einen gewissen Zeitraum zur Verfügung zu stellen. Dies kann in pessimistischer Form („Flugticket wird für 2 Stunden provisorisch gebucht. Wenn bis dahin keine Bestätigung kommt, erlischt die Reservierung“) oder optimistischer Form („Hotelzimmer ist gebucht. Eine Stornierung ist bis 24h vor Ankunft möglich“) erfolgen. Dadurch ergeben sich Vorteile sowohl auf Seiten des Koordinators als auch auf Seiten des Teilnehmers:

- Koordinator: Genaue Informationen über den Status des(der) Teilnehmer(s) und somit die Möglichkeit dementsprechend die Handlungen zu setzen
- Teilnehmer: keine unnötige Blockade von Ressourcen und dadurch Möglichkeit zur Performanceerhöhung

Weiters ist es dem Koordinator durch das open top completion protocol möglich, die Zeit zwischen den beiden Phasen des 2PO selbst zu bestimmen, was ihm im Vergleich zum 2PC-Protokoll eine wesentlich höhere Flexibilität verleiht.

Weniger förderlich für die Umsetzung des Holiday Packages, das ja auch in heterogenen Umgebungen funktionieren soll, ist das bereits vorhin angesprochene Nichtvorhandensein einer Transaktionsprotokollspezifikation. Zwar gibt es mit dem spezifizierten SOAP/HTTP Binding eine geeignete Möglichkeit, das BTP für Web Services zu verwenden, doch solange dieses nicht Standard ist, gibt es keine Garantie für die so wichtige Interoperabilität.

Neben dem Transportprotokoll-Problem ist auch die Mischung von Applikations- und Transaktionslogik (wie am Beispiel der Qualifier zu sehen) für die Implementierung eher wenig förderlich. Weiters reduziert die Verwendung von nicht-standard Qualifiern, wie sie von BTP zugelassen sind, die Applikationsportabilität (sinngemäß nach [Litt04]).

Für das Holiday Package zwar ausreichend, sind die Transaktionsmöglichkeiten beim BTP nur auf Atoms und Cohesions sowie Interposition beschränkt. Die Umsetzung von sehr komplexen Geschäftsprozessen mit BTP könnte sich daher als nicht ganz trivial erweisen. Auch könnte die Tatsache, dass das im BTP verwendete open top completion protocol die ACID-Semantik nicht impliziert, Probleme bei der Verarbeitung von Transaktionen (Transaktionsteilen), die diese Semantik benötigen, mit sich bringen.

Schlussendlich kann man bezogen auf das Holiday Package sagen, dass das sehr umfangreiche BTP zwar grundsätzlich für die Umsetzung geeignet wäre, es allerdings doch einige entscheidende Schwachpunkte gibt.

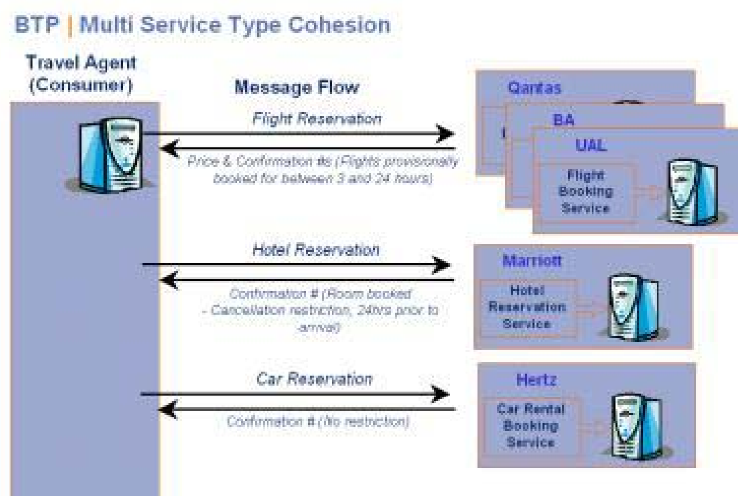


Abbildung 3.6: BTP Multi Service Type Cohesion (aus [Pott02])

Wie die Umsetzung des Holiday Packages mit BTP aussehen kann, ist grob in Abbildung 3.6 zu sehen. Diese stellt lediglich einen Auszug aus [Pott02] dar. In [Pott02] sind mehrere komplette Beispiele enthalten, die hier aus Platzgründen nicht abgebildet werden können, die aber sehr intuitiv sind und das Konzept von BTP sehr gut darstellen und deshalb der interessierten LeserIn ans Herz gelegt werden.

3.3 WS-Coordination und WS-Transaction

Neben dem BTP gibt es noch weitere Möglichkeiten zur Koordination von Transaktionen mit Web Services. Eine davon ist die Verwendung der Protokolle WS-Coordination (in weiterer Folge WS-C genannt) und WS-Transaction (WS-Tx), deren erste Spezifikationen im August 2002 von IBM, Microsoft und BEA veröffentlicht wurde. Während der Jahre 2003 und 2004 wurde die WS-T Spezifikation in die Protokolle WS-AtomicTransaction (WS-AT) und WS-BusinessActivity (WS-BA) aufgespaltet.

Die in dieser Arbeit herangezogenen Spezifikationen sind [Cabr04a], [Cabr04b] und [Cabr04c]. Alle drei stammen vom November 2004. Des Weiteren wurde [Litt04] für die Erstellung der Inhalte dieses Unterkapitels herangezogen.

3.3.1 Web Services Coordination

WS-Coordination stellt ein erweiterbares Framework zur Koordination von Aktivitäten dar und dient verteilten Systemen dazu, unabhängig ihrer proprietären Protokolle eine Koordination untereinander zu erzielen. WS-Coordination greift dabei auf SOAP und WSDL zurück (vgl. Unterkapitel 3.1) und besteht im Wesentlichen aus drei Komponenten:

- Aktivierungsservice (Activation Service)
- Registrierungsservice (Registration Service)
- Menge an Koordinationsprotokollen (spezifiziert durch den eingesetzten Koordinationstyp; hier Protokollservice genannt)

Letztere Komponente stellt die wesentliche Erweiterungsmöglichkeit dar, da verschiedene Koordinationstypen (und damit verbundene Protokolle) wie Plug-Ins eingesetzt und so das WS-Coordination Framework an individuelle Anforderungen angepasst werden kann (beispielsweise verschiedene Transaktionstypen und Protokolle – siehe später WS-AT und WS-BA; oder WS-Security etc.)

Das Aktivierungsservice stellt eine Möglichkeit zur Erzeugung eines neuen Koordinationskontexts dar. Im Normalfall sendet der Initiator/die Initiatorapplikation eine, wie in Listing 3.2 beispielhaft dargestellte, CreateCoordinationContext-Nachricht an das Aktivierungsservice. In dieser Nachricht enthalten sein muss der gewünschte Koordinationstyp (in Listing 3.2 wurde WS-AT als Koordinationstyp gewählt).

Listing 3.2:

```
<CreateCoordinationContext>
     <CoordinationType>
          http://schemas.xmlsoap.org/ws/2004/10/wsat
     </CoordinationType>
</CreateCoordinationContext>
```

Des Weiteren ist es möglich, einen bisherigen Kontext (Element `<CurrentContext>`) anzugeben. Dadurch wird interposition ermöglicht. Zusätzlich bleibt zur Angabe weiterer Informationen die Möglichkeit, beliebige Elemente in die Anfrage einzubauen sowie der Einsatz des `<Expires>`-Elements, das die Gültigkeitsdauer des Kontexts definiert.

Als Antwort auf die CreateCoordinationContext-Nachricht sendet das Aktivierungsservice eine CreateCoordinationContextResponse-Nachricht mit dem Kontext (Element `<CoordinationContext>`) an den Initiator. Enthalten im Kontext ist im Normalfall neben einem Kontext-Identifier der Koordinationstyp sowie Informationen über das Registrierungsservice (Element `<RegistrationService>`) wie beispielsweise die Adresse dieses.

Der dem Initiator sodann zur Verfügung stehende Kontext wird an den(die) Teilnehmer des Kontexts geschickt (z.B. im SOAP Header des Web Service Aufrufs). Der Initiator kann auch selbst Teilnehmer am Kontext sein (das Senden des Kontexts an sich selbst entfällt selbstverständlich).

Die dem Teilnehmer durch den Kontext zur Verfügung stehenden Informationen (Koordinationstyp, Adresse des Registrierungsservice) werden von ihm genutzt, um sich für eines oder mehrere durch den Koordinationstyp zur Verfügung stehende Protokolle durch Angabe des ProtocolIdentifier beim Registrierungsservice zu registrieren (Nachricht in abstrakter Form in Listing 3.3 dargestellt). Wichtig bei der Registrierung ist die Angabe des `<ParticipantProtocolService>`, das den Protokoll-Endpunkt des Teilnehmers angibt, um eine spätere Kommunikation zwischen Protokoll-Service und Teilnehmer zu ermöglichen. Form des Endpoints ist in der WS-Adressing Spezifikation ([Box04]) definiert.

Listing 3.3:

```
<Register>
    <ProtocolIdentifier>
        http://schemas.xmlsoap.org/ws/2004/10/wsat/Durable2PC
    </ProtocolIdentifier>
    <ParticipantProtocolService>
        <!-- Form definiert in WS-Adressing ([Box04]) -->
    </ParticipantProtocolService>
    <!-- weitere Elemente für zusätzliche Informationen -->
</CreateCoordinationContext>
```

Die Antwort des Registrierungsservice (RegistrationResponse) enthält den Endpunkt des Koordinators (`<CoordinationProtocolService>`).

Ein Teilnehmer kann sich durch mehrmaliges Senden von Nachrichten an das Registrierungsservice auch für mehrere Protokolle gleichzeitig registrieren.

Unter dem Begriff Koordinator (oder CoordinationService) wird die Aggregation von Aktivierungs-, Registrierungs- und Protokollservice verstanden. Das Aktivierungsservice kann vom Koordinator unterstützt werden, das Registrierungsservice muss ([Cabr04a]).

In Abbildung 3.7 ist der Ablauf der Aktivierung und Registrierung in Sequenzdiagramm-Form exemplarisch dargestellt. Die Protokollnachrichten, die nach Registrierung zwischen dem Protocol Service und dem(den) Protokollteilnehmer(n) gesendet werden sind nicht Teil von WS-Coordination, sondern protokollabhängig und daher in Abbildung 3.7 nur abstrakt eingezeichnet.

Die Kommunikation zwischen Teilnehmer und Koordinator x erfolgt oftmals indirekt (Teilnehmer benutzt einen Koordinator y zur Kommunikation mit x). Dies kann beispielsweise aus Vertrauens- oder Performanzgründen der Fall sein.

Auf das Security-Modell und andere Details die von Cabrera et al. in [Cabr04a] vorgestellt werden und enthalten sind, wird im Rahmen dieser Arbeit nicht näher eingegangen. Die interessierte LeserIn sei daher darauf verwiesen.

Zur Verarbeitung von Transaktionen unter der Verwendung des WS-Coordination Frameworks gibt es zwei Koordinationstypen:

- Web Service Atomic Transaction (WS-AtomicTransaction) und
- Web Service BusinessActivity Framework (WS-BusinessActivity)

Während WS-AtomicTransaction für kurze (klassische) ACID Transaktionen gedacht ist, zielt WS-BusinessActivity auf langlebige Transaktionsformen, wie sie für Business Activities benötigt werden, ab.

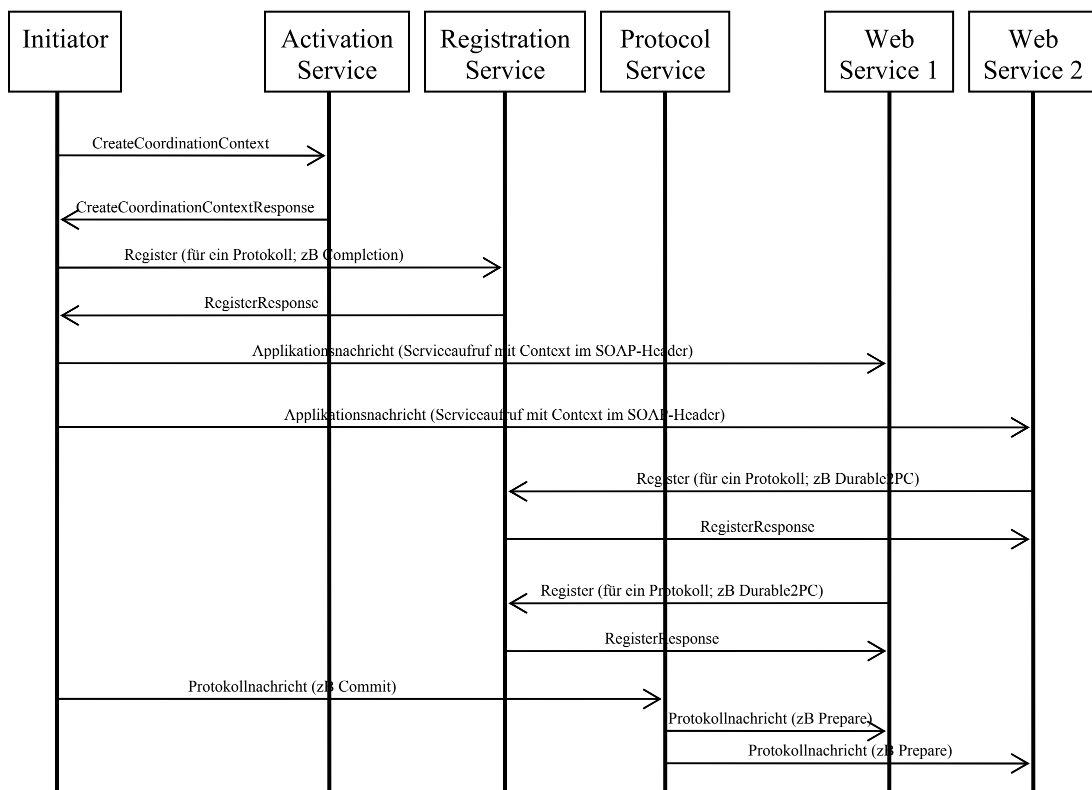


Abbildung 3.7: Beispielhafter Ablauf von Aktivierung und Registrierung in WS-Coordination

3.3.2 Web Services Atomic Transaction

Um kurze Transaktionen mit atomarem Ausgang durchführen zu können, stellt der Koordinationstyp WS-AT drei Protokolle zur Verfügung. Bevor diese angewendet werden können, muss – wie bereits bei WS-C erklärt – zuerst ein entsprechender Koordinationskontext erzeugt werden. Dazu muss beim Senden der CreateCoordinationContext-Nachricht der Koordinationstyp WS-AT angegeben werden. Dieser wird mittels des Identifiers

`http://schemas.xmlsoap.org/ws/2004/10/wsat`

identifiziert. Wird weiters bei der Erzeugung das CurrentContext-Element inkludiert, so wird der neu erzeugte Koordinator als Nachfolger des im CurrentContext-Element angegebenen Koordinator behandelt. Dadurch ist es möglich, Hierarchien aufzubauen. Ist das CurrentContext-Element nicht enthalten, erzeugt der Koordination eine neue Transaktion und fungiert als Root.

Weiters kann der CoordinationContext ein Expires-Attribut enthalten, das den frühesten Zeitpunkt definiert, an dem die Transaktion automatisch terminieren kann.

Der vom Aktivierungsservice anschließend erhaltene Transaktionskontext muss in allen Applikationsnachrichten enthalten sein.

Die drei für die Registrierung für die Teilnehmer zur Verfügung stehenden Protokolle sind:

- Completion Protokoll
- Volatile 2PC Protokoll
- Durable 2PC Protokoll

Completion Protokoll

Das Completion Protokoll wird vom Initiator zum Beenden der Transaktion verwendet (Initiator teilt dem Koordinator mit, ob die Transaktion mit Commit oder Rollback abgeschlossen werden soll). Um sich für das Protokoll zu registrieren, muss folgender Identifier verwendet werden:

```
http://schemas.xmlsoap.org/ws/2004/10/wsat/Completion
```

Abbildung 3.8 (aus [Cabr04b]) veranschaulicht das Protokoll.

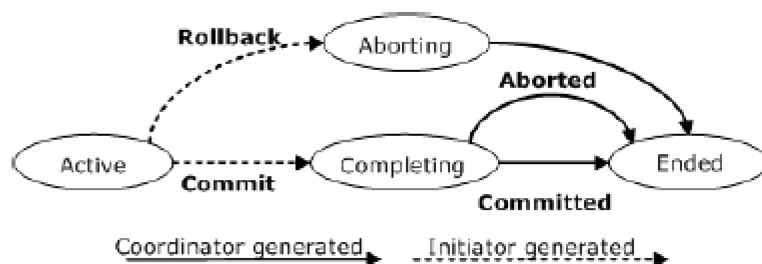


Abbildung 3.8: WS-AT Completion Protokoll (aus [Cabr04b])

Volatile und Durable Two-Phase Commit

Zum Erreichen eines atomaren Ergebnisses unter den Teilnehmern einer WS-AT Aktivität wird das bereits in Kapitel 2 besprochene Two-Phase Commit (2PC) Protokoll verwendet. Um Applikationen im Web, die sehr häufig Zwischenspeicher (Caches) verwenden, Rechnung zu tragen, liegt das 2PC Protokoll in WS-AT in zwei Varianten vor: Volatile 2PC und Durable 2PC.

Während das Volatile 2PC Protokoll für Ressourcen wie Caches gedacht ist, sind die „Zielgruppe“ des Durable 2PC Protokolls Teilnehmer mit dauerhaften Ressourcen (wie Datenbanken). Ein Teilnehmer kann sich allerdings für mehr als eines dieser beiden Protokolle registrieren. Die Identifier dafür sind

```
http://schemas.xmlsoap.org/ws/2004/10/wsat/Volatile2PC
```

beziehungsweise

`http://schemas.xmlsoap.org/ws/2004/10/wsat/Durable2PC`

Abbildung 3.9 aus [Cabr04b] zeigt den Ablauf des in WS-AT verwendeten 2PC Protokolls. ReadOnly in der Abbildung wird vom Teilnehmer gesendet, wenn er an der Transaktion nur lesend beteiligt ist (war). Er signalisiert damit, dass er am 2PC nicht teilnehmen möchte, da dafür ja kein Bedarf besteht.

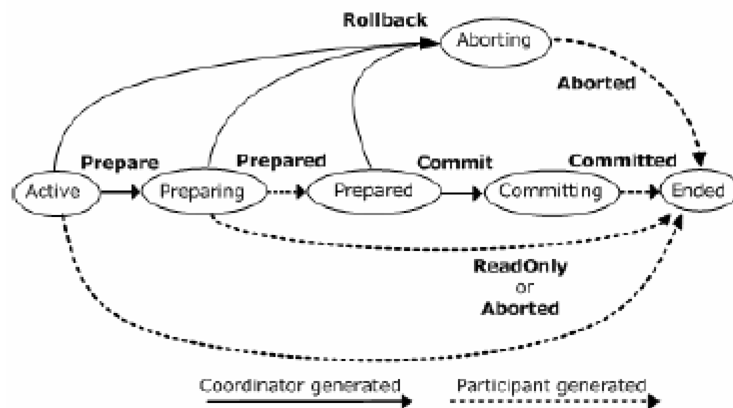


Abbildung 3.9: Two-Phase Commit Protokoll in WS-AT (aus [Cabr04b])

Nachdem der Initiator der Transaktion Commit (via zuvor vorgestelltem Completion Protokoll) an den Root Koordinator gesendet hat, beginnt dieser mit der Prepare Phase für alle für den Volatile 2PC registrierte Teilnehmer. Erst nach Erhalt einer Antwort von jedem dieser Teilnehmer kann der Koordinator Prepare an die Teilnehmer des Durable 2PC senden. Dabei ist zu beachten, dass Prepare nur dann an die Teilnehmer des Durable 2PC gesendet wird, wenn die Prepare Phase des Volatile 2PC erfolgreich abgeschlossen wurde. Bevor nun Commit an die Teilnehmer der beiden Protokolle gesendet werden kann, müssen die Teilnehmer des Durable 2PC mit Prepared oder ReadOnly antworten.

Da es nur zu einer Notifikation der Teilnehmer kommt, wenn es zu einem Prepare an die Teilnehmer des Durable 2PC kommt, gibt es für die Teilnehmer am Volatile 2PC keine Garantie, das Ergebnis der Transaktion zu erfahren.

Teilnehmern ist es solange möglich, sich für einen Koordinator zu registrieren, bis der Koordinator Prepare an einen (beliebigen) Teilnehmer des Durable 2PC Protokolls sendet ([Cabr04b]).

Durch die Unterscheidung von Volatile und Durable 2PC wird die Möglichkeit geschaffen, eine Reihung von Aufrufen verschiedener Teilnehmertypen vorzunehmen, was Performanceoptimierungen erlaubt. Teilnehmer die beispielsweise Caches ver-

wenden, können durch die Registrierung für das Volatile 2PC Protokoll früher von der bevorstehenden Beendigung der Transaktion erfahren und Daten im Cache vor der Prepare Phase des Durable 2PC rechtzeitig in die Datenbank zurückschreiben, ohne dabei Teilnehmer des Durable 2PC „aufzuhalten".

3.3.3 Web Services Business Activity Framework

WS-BA wurde – wie eingangs erwähnt – konzipiert, um lange Transaktionen beziehungsweise die Abbildung von Geschäftsprozessen mit komplexen Strukturen zu ermöglichen.

Eine BusinessActivity besteht aus (einer Reihe von) so genannten BusinessActivity scopes (Ein scope stellt einen Business Task dar).

Wie bei den in Kapitel 2 vorgestellten geschachtelten Transaktionen ist auch bei WS-BA eine Schachtelung und Hierarchiebildung der scopes jeglicher Tiefe möglich.

Für den WS-BA CoordinationContext stehen zwei Koordinationstypen zur Verfügung; nämlich AtomicOutcome und MixedOutcome, die durch die URIs

`http://schemas.xmlsoap.org/ws/2004/10/wsba/AtomicOutcome`

beziehungsweise

`http://schemas.xmlsoap.org/ws/2004/wsba/MixedOutcome`

identifiziert werden. Während beim AtomicOutcome-Koordinationstyp ein atomares Ergebnis unter den Teilnehmern eines scopes (bzw. der BA) notwendig ist, steht es dem Koordinator beim Typ des MixedOutcome für jeden einzelnen Teilnehmer frei, wie dessen Ausgang ist. Das Ergebnis kann beim MixedOutcome atomar sein, muss aber nicht. Der Spezifikation [Cabr04c] zufolge müssen alle Koordinatoren den Koordinationstyp des AtomicOutcome implementieren. Der Koordinationstyp MixedOutcome kann implementiert werden.

Um zu einem Ergebnis unter den Teilnehmern zu kommen, stellen Cabrera et al. in [Cabr04c] zwei Koordinationsprotokolle zur Verfügung:

- BusinessAgreementWithParticipantCompletion
- BusinessAgreementWithCoordinatorCompletion

Die Registrierung für eines dieser beiden Protokolle erfolgt über die Protokoll-Identifier

`http://schemas.xmlsoap.org/ws/2004/10/wsba/ParticipantCompletion`

beziehungsweise

`http://schemas.xmlsoap.org/ws/2004/10/wsba/CoordinatorCompletion`

Sowohl ParticipantCompletion als auch CoordinatorCompletion verwenden das Konzept der Kompensation (siehe Kapitel 2) in der Art, als dass die einzelnen Teilnehmer an sie gestellten Anfragen vorzeitig permanent durchführen und deren Effekte gegebenenfalls durch kompensierende Operationen wieder ausgleichen. Wie die Kompensation durchgeführt wird, ist nicht Teil von WS-BA, sondern liegt beim Teilnehmer. Weiters ist es jedem Teilnehmer möglich, die BusinessActivity zu verlassen. Dies führt dazu, dass die Teilnehmeranzahl während der BusinessActivity dynamisch ist, was bei der Abarbeitung eines Geschäftsprozesses alles andere als außergewöhnlich ist.

Wie auch schon bei WS-AT ist es auch beim WS-BA CoordinationContext möglich, ein `<Expires>`-Attribut zu verwenden, das den frühesten Zeitpunkt, an dem eine Aktivität eigenmächtig terminiert werden kann, festlegt.

Beide Protokolle spezifizieren das gleiche Verhalten zwischen Koordinator und Teilnehmer, mit dem einzigen Unterschied, dass beim ParticipantCompletion-Protokoll die Initiative von den Teilnehmern ausgeht, während beim Coordinator-Completion-Protokoll die Teilnehmer vom Koordinator abhängig sind. Zum Vergleich sind beide Protokolle in den Abbildung 3.10 und Abbildung 3.11 (beide aus [Cabr04c]) dargestellt.

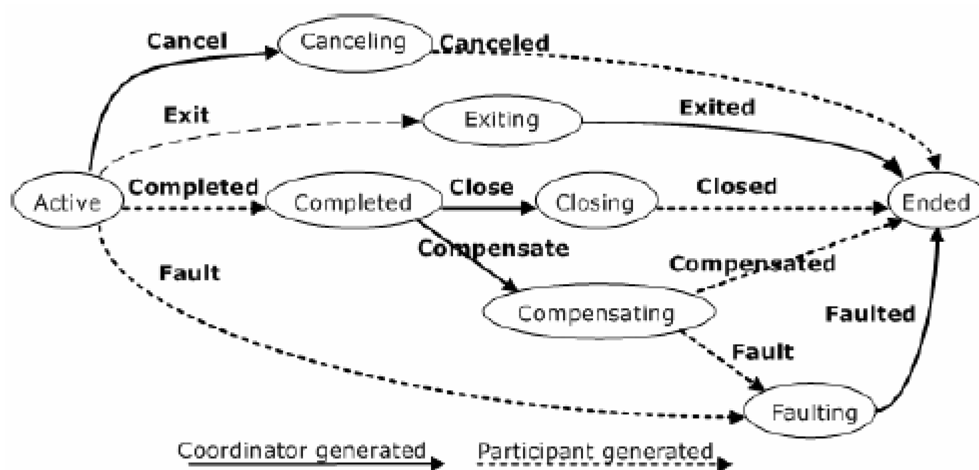


Abbildung 3.10: BusinessAgreementWithParticipantCompletion Protokoll

Wie in Abbildung 3.10 zu sehen, behandelt jeder Teilnehmer die an ihn gestellte Anfrage und teilt seinen Ausgang dem Koordinator mit. Kann die Operation vom Teilnehmer nicht durchgeführt werden oder möchte er aus der Activity aussteigen, so informiert er (mittels Fault beziehungsweise Exit) den Koordinator und wartet nur noch auf dessen Bestätigung. Möchte er hingegen an der Aktivität weiterhin teilnehmen (erfolgreiche Durchführung vorausgesetzt), wartet er auf das Ergebnis des Koordinators und muss gegebenenfalls (bei Erhalt von Compensate) eine kompensierende Operation durchführen.

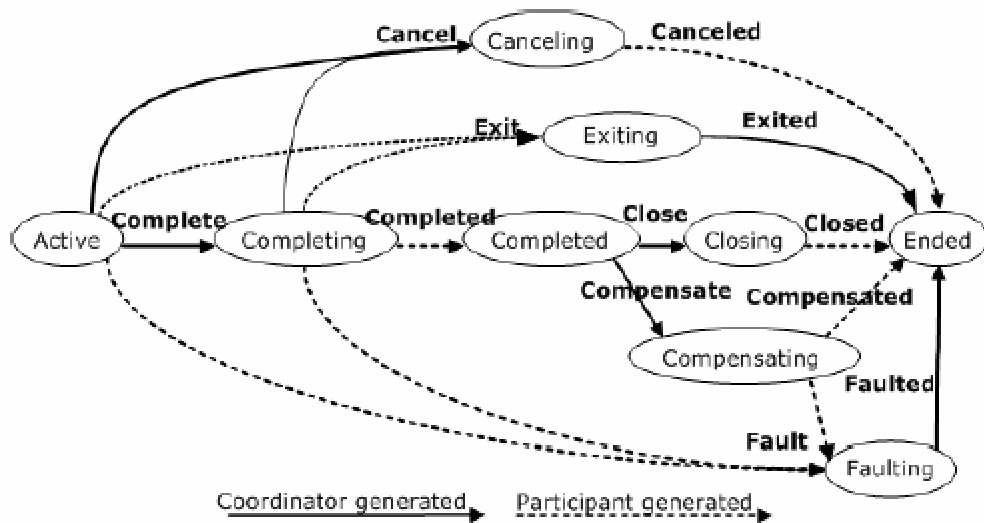


Abbildung 3.11: BusinessAgreementWithCoordinatorCompletion Protokoll

Der im Vergleich zu Abbildung 3.10 in Abbildung 3.11 zu sehende Unterschied ist, dass der Koordinator eine Complete Nachricht an die Teilnehmer schickt und ihnen somit signalisiert, dass sie keine weiteren Anfragen erhalten und die Verarbeitung abschließen sollen.

Der große Vorteil der ParticipantCompletion gegenüber der Variante der CoordinatorCompletion ist der, dass bei Auftreten eines Fehlers/gewünschten Ausstiegs des Teilnehmers der Koordinator sofort informiert wird und die Applikation dem Geschäftsprozess entsprechende Handlungen setzen kann (zum Beispiel neue Teilnehmer suchen etc.) und nicht erst, wie bei der CoordinatorCompletion, bei Beendigung der Aktivität von der Nichtdurchführbarkeit/dem Nichtdurchführungswunsch in Kenntnis gesetzt wird.

Anders als es beim Completion Protokoll für WS-AT der Fall ist, ist die Kommunikation zwischen Initiatorapplikation und Koordinator zum Beenden der Transaktion für WS-BA nicht spezifiziert. Daher liegt es nahe, dass, auch aufgrund der Tatsache,

dass bei BusinessActivities die Einbindung der Anwendungslogik in die Koordination nötig ist, Applikation und Koordinator eng zusammenarbeiten beziehungsweise die Initiatorapplikation gleichzeitig als Koordinator fungiert, um den Ausgang der BusinessActivity steuern zu können. Eine weitere Möglichkeit wäre, die in [Cabr04c] betonte Erweiterbarkeit von WS-Coordination zu nutzen und beispielsweise ein eigenes Kommunikationsprotokoll zwischen Initiator und Koordinator zu definieren oder spezielle Verhaltensregeln des Koordinators festzulegen.

Neben den bisher genannten Teilen enthält die Spezifikation noch Statustabellen für Teilnehmer und Koordinator, in denen abzulesen ist, wie bei welchem Status auf welche Nachrichten reagiert wird. Auch die allgemein bei WS-Coordination angesprochene Sicherheit wird für den Einsatz von WS-BA empfohlen. Beide Teile können in [Carb04b] nachgelesen werden.

3.3.4 WS-Tx und Holiday Package

Durch die von den Protokollen WS-AT und WS-BA angebotenen Koordinationstypen ist es möglich, sowohl die nach wie vor wichtigen und weit verbreiteten kurzen ACID Transaktionen in verteilten, heterogenen Systemen durchzuführen, als auch, vor allem für das Holiday Package benötigte, lange Transaktionen zu verarbeiten.

Beim WS-AT Protokoll ist vor allem die Möglichkeit des Volatile 2PC, der eine Möglichkeit zur Synchronisation bietet und damit den Einsatz von Caching erleichtert, hervorzuheben. Auch wenn WS-AT für das Holiday Package eher weniger geeignet ist, da die Erfüllung der ACID-Kriterien in diesem Fall nicht zielführend ist, ist es dennoch bedeutsam, ein derartiges Protokoll zur Verfügung zu haben, falls ACID von Nöten ist.

WS-BA hingegen bietet mit der Einteilung einer BusinessActivity in scopes, dynamischen Teilnehmerlisten und Schachtelungsmöglichkeiten ideale Voraussetzungen für die Modellierung des Holiday Packages. Mit diesen Eigenschaften und der Möglichkeit der Applikation, entscheiden zu können, welche scopes schlussendlich für die erfolgreiche Durchführung der BusinessActivity herangezogen werden können beispielsweise alternative Flugrouten als ein Task gesucht werden (von denen schlussendlich nur die billigste Variante gebucht wird), während die Hotelbuchung und die Reservierung eines Mietautos in anderen Tasks ablaufen. Auch kann dynamisch auf Fehler oder Exits von Teilnehmern durch Aufnahme neuer Teilnehmer in die BusinessActivity reagiert werden (z.B. sind zwei der drei zum Preisvergleich gewählten Airlines ausgebucht und die dritte ist wegen Netzwerkproblemen plötzlich

nicht mehr erreichbar; es werden sodann einfach neue Airlines zum Preisvergleich ausgewählt). Auch ein Nichterhalten eines Mietautos, das in vielen Fällen keinen vitalen Bestandteil einer Reise darstellt, würde durch die Lockerung der Atomizität zu keinem Abbruch der BusinessActivity führen.

Anders als beim BTP, bei dem lediglich Interposition möglich ist, ist bei der Aufteilung einer BusinessActivity in scopes, die ja wie geschachtelte Transaktionen funktionieren, bei der Nichtdurchführbarkeit eines scopes nicht die gesamte Transaktion betroffen.

Weiters führt die in WS-BA durch vorzeitige Sichtbarmachung von Ergebnissen (gemeinsam mit Kompensation), die eine Lockerung des Isolationskriteriums bedeutet, zur Erhöhung der Nebenläufigkeit und eröffnet damit den Weg für lange Transaktionen.

Ein weiterer gelungener Punkt von WS-BA hinsichtlich des Holiday Packages ist die teilnehmergesteuerte Beendigung eines scopes (BusinessAgreementWithParticipantCompletion Protokoll), durch die zum Einen jeder Teilnehmer rasch die angeforderte(n) Operation(en) vollständig durchführen kann, ohne den „Befehl" des Koordinators abwarten zu müssen und zum Anderen Fehler oder Ausstieg dem Koordinator unmittelbar mitgeteilt werden können ohne den Abschluss der Transaktion abwarten zu müssen.

Vorteilhaft für die einzelnen Teilnehmer ist auch, dass sie aus den BusinessActivities aussteigen uns so ihren eigenen Bedürfnissen optimal nachkommen können (Reserviert beispielsweise Applikation x einen Platz für Flug y, in dem noch 10 Plätze frei sind und anschließend bucht Applikation z 10 Plätze für Flug y, so ist es für die Fluggesellschaft wohl besser, wenn sie aus der BusinessActivity von x aussteigen kann und stattdessen die Buchung von y ausführt).

Auch aus technischer Sicht bietet WS-Tx (WS-AT und WS-BA) eine gute Grundlage für Transaktionen mit Web Services. Während beim BTP die Bindung an HTTP und SOAP freigestellt ist, ist diese im WS-AT und WS-BA zugrunde liegenden WS-Coordination Framework vorausgesetzt, was für die Interoperabilität bei Web Services sicher keinen Nachteil darstellt. Schließlich wurde WS-C, wie dessen Name schon sagt, speziell für Web Services entworfen. Durch die Verwendung von WS-Coordination ist es auch relativ einfach, Adaptionen an den Protokollen zu machen oder bei Bedarf neue Protokolle zu spezifizieren und einzubinden, ohne die vorhandene Architektur grundlegend ändern zu müssen. Auch der Sicherheitsaspekt, auf

den hier nicht näher eingegangen wird, wurde in WS-C, WS-AT und WS-BA bedacht und ist gerade bei verteilten Anwendungen ein wichtiger Bestandteil.

Kritisch zu betrachten ist die einerseits positive Möglichkeit der vorzeitigen Sichtbarmachung von Operationen in Kombination mit dem MixedOutcome Koordinationstyp bei WS-BA. Man stelle sich dazu folgendes Szenario vor: Der Koordinator bucht je einen Platz für drei Flüge und entscheidet sich dann für einen und die anderen beiden werden zur Kompensation aufgefordert (was ja mittels WS-BA möglich ist). Dies ist zwar auf den ersten Blick nicht weiter tragisch, problematisch wird es allerdings, wenn beispielsweise der Koordinator zehn Flüge heranzieht und davon schlussendlich nur einen bucht. Nutzen nun viele User die Flugbuchung, sind für die Dauer der BusinessActivity (bzw. des scope) im Worst Case 100% aller Plätze gebucht, obwohl eigentlich nur 10% der Plätze benötigt werden (bei drei Flügen wäre es entsprechend 1/3, was allerdings auch suboptimal ist). Das Problem muss natürlich nicht auftreten, da es sehr stark von der Applikationslogik abhängt, doch sollte es beim Entwurf dieser Beachtung finden.

Betrachtet man die Spezifikationen von WS-AT ([Cabr04b]) und von WS-BA ([Cabr04c]) an sich, so fällt auf, dass WS-BA im Vergleich zu WS-AT kein Protokoll zur Kommunikation zwischen Initiatorapplikation und Koordinator bereitstellt. Cabrera et al. stellen zwar in [Cabr04c] den Anspruch, dass die Applikation entscheiden kann, welche child tasks schlussendlich ausgeführt werden (genauer Wortlaut: „Allow a business application to select which child tasks are included in the overall outcome processing.“), das WIE geht aus der Spezifikation allerdings nicht genau hervor. Der Grund dafür ist höchstwahrscheinlich die individuell unterschiedliche Komplexität von Geschäftsprozessen, die jeweils eine spezifische Lösung benötigt. Das Fehlen des WIE erfordert individuelle Lösungen durch bereits erwähnte Verschmelzung von Koordinator und Applikation oder Entwurf eigener Kommunikationsprotokolle. In Zeiten der immer stärkeren Modularisierung und Serviceorientierung wäre es dennoch wünschenswert, wenn ein allgemeines Kommunikationsprotokoll ähnlich dem Completion Protokoll in WS-AT in die Spezifikation von WS-BA aufgenommen würde.

Nichtsdestotrotz stehen mit WS-C, WS-AT und WS-BA geeignete Frameworks, Protokolle und Koordinationstypen zur Verfügung, um das Holiday Package Beispiel im Rahmen von Web Services adäquat durchführen zu können.

3.4 WS-Composite Application Framework

Drittes und letztes in dieser Arbeit behandeltes Protokoll für Transaktionen mit Web Services ist das Web Services Composite Application Framework, das im Juli 2003 von Arjuna Technologies, Fujitsu, IONA Technologies, Oracle und Sun entwickelt wurde. Die Ziele dafür waren die Unterstützung verschiedenster Transaktionsmodelle sowie die Zusammenarbeit mit existierenden Systemen wie etwa WS-Security.

WS-CAF besteht aus drei Teilen, nämlich

- Web Services Context (WS-CTX),
- Web Services Coordination Framework (WS-CF) und
- Web Services Transaction Management (WS-TXM)

und wurde bereits bei OASIS zur Standardisierung eingereicht. Derzeitiger Stand: es gibt bereits erste Drafts von WS-CTX und WS-CF; WS-TXM wurde im Mai 2005 in drei Teile aufgeteilt (WS-ACID, WS-LRA und WS-BP), die im Moment den noch später in diesem Unterkapitel behandelten Teilen von WS-TXM 1:1 entsprechen, da die Arbeiten daran erst begonnen werden. Der aktuelle Entwicklungsstand kann auf

```
http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ws-caf
```

verfolgt werden. Da sich WS-CAF gerade im Umbruch befindet, wurden die aktuellen Versionen [Bunt03a], [Bunt03b] und [Bunt03c] - auch auf Anraten des Secretaries des technischen Kommittees für WS-CAF, Dr. Mark Little, - für die entsprechenden Teile dieses Unterkapitels herangezogen. Auch [Litt04] sowie [Bunt03d] dienten bei der Erstellung der Inhalte.

3.4.1 Zusammenhang zwischen WS-CTX, WS-CF und WS-TXM

Um Web Services miteinander in Beziehung zu setzen bietet das WS-CAF durch das Layering der drei Teile WS-CTX, WS-CF und WS-TXM (siehe Abbildung 3.12) die Möglichkeit des einfachen Teilens eines gemeinsamen Kontext bis hin zur genauen Abstimmung der Handlungen der beteiligten Web Services und damit eine gute Grundlage zur Variierung für unterschiedliche Ansprüche.

WS-CTX, das ein leichtgewichtiges Kontextmanagementsystem darstellt, kann unabhängig von WS-CF und WS-TXM verwendet werden und bietet den beteiligten Web Services einen einfachen Mechanismus, einen gemeinsamen Kontext zu definieren beziehungsweise gemeinsame Informationen austauschen zu können. Der Kontext, der mindestens eine ID enthalten muss und applikationsabhängig ist, wird

als Web Ressource dargestellt und kann entweder über einen URI im Header referenziert oder als Ganzes im SOAP Body einer Nachricht übermittelt werden.

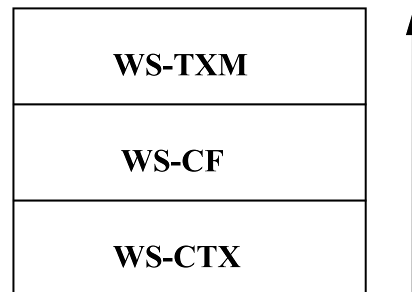


Abbildung 3.12: Schichten von WS-CAF

Auf WS-CTX aufbauend geht WS-CF einen Schritt weiter und definiert einen Koordinator, der für die Benachrichtigung aller dem gleichen Kontext zugehörigen Web Services verantwortlich ist. WS-CF ist vergleichbar mit WS-Coordination. Auch hier registrieren sich die Teilnehmer beim Koordinator, der in weiterer Folge die Koordination der Teilnehmer und das Propagieren von Ergebnissen übernimmt. Ein Koordinator selbst kann sich selbstverständlich bei einem anderen Koordinator registrieren, wodurch Interposition möglich wird. WS-CF, das unabhängig von der WS-TXM Spezifikation eingesetzt werden kann, dient diesem wiederum als Basis. Während WS-CF nur für die Notifikation verantwortlich ist, definiert die WS-TXM Spezifikation verschiedene Protokolle zur Abstimmung gemeinsamer transaktionaler Aktivitäten unter den Teilnehmern.

In Abbildung 3.13 (aus [Bunt03d]) ist der Zusammenhang zwischen WS-CTX, WS-CF und WS-TXM grafisch dargestellt. Details zu WS-CTX und WS-CF, die weitaus umfangreicher sind, als hier vorgestellt, werden im Gegensatz zu WS-TXM aus Relevanzgründen in dieser Arbeit nicht weiter behandelt und können in den jeweiligen Spezifikationen [Bunt03a] beziehungsweise [Bunt03b] im Detail nachgelesen werden.

WS-TXM umfasst drei Transaktionsprotokolle, wodurch unterschiedliche Transaktionsmodelle und Architekturen unterstützt werden und eine genaue Abstimmung auf transaktionale Bedürfnisse möglich wird. Diese in weiterer Folge noch näher vorgestellten Protokolle sind:

- *ACID Transactions* für kurze Transaktionen, die den ACID Kriterien genügen müssen
- *Long Running Action* für lange Transaktionen; Isolation aufgeweicht

- *Business Process Transactions* für komplexe Transaktionen zur Darstellung und Verarbeitung von umfangreichen Geschäftsprozessen

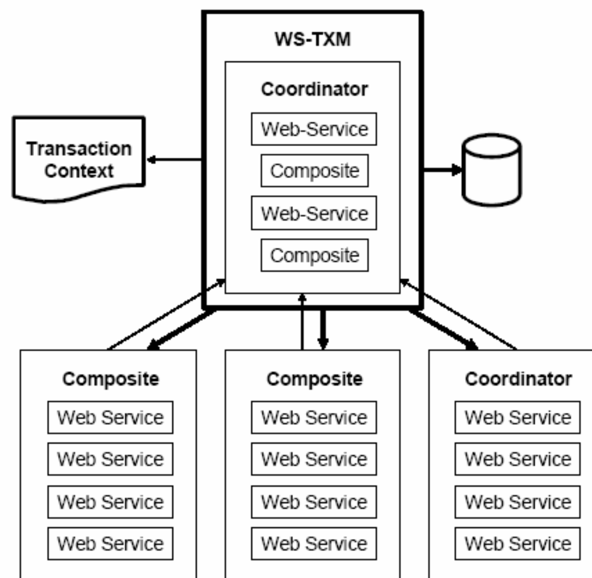


Abbildung 3.13: Zusammenhang zwischen WS-CTX, WS-CF und WS-TXM (aus [Bunt03d])

3.4.2 ACID Transactions

Der oftmals notwendige und für die Interoperabilität mit existierenden Systemen entworfene Koordinationstyp ACID Transactions (AT; neu: WS-ACID) ähnelt sehr dem von WS-Tx zur Verfügung gestellten WS-AT Protokoll. Es werden daher nur die wichtigsten Teile erwähnt.

Das ACID Transaction Modell wird durch den URI

```
http://www.webservicestransactions.org/wsdl/wstxm/tx-
acid/2003/03
```

identifiziert und stellt wie auch WS-AT zwei Sub-Protokolle zur Erreichung eines atomaren Ergebnisses unter den Teilnehmern zur Verfügung: ein traditionelles 2PC- sowie das *synchronization*-Protokoll. Die URIs dafür sind

```
http://www.webservicestransactions.org/wsdl/wsTXM/tx-
acid/2pc/2003/03
```

beziehungsweise

```
http://www.webservicestransactions.org/wsdl/wsTXM/tx-
acid/sync/2003/03
```

Den Teilnehmern des 2PC ist es nach erfolgreicher Prepare-Phase möglich, eine autonome Entscheidung darüber zu treffen, ob sie mit Commit oder Rollback abschließen. Abhängig vom Ausgang der Transaktion, über den die Teilnehmer schlussend-

lich vom Koordinator informiert werden, kann es durch die eigenmächtige Entscheidung der Teilnehmer zu Inkonsistenzen kommen (z.B. Teilnehmer commited, aber Koordinator erteilt Rollback). Dieser mögliche heuristische Ausgang (heuristic outcome) muss dem Koordinator mitgeteilt werden.

Wie bereits beim WS-AT Protokoll kann ein Teilnehmer durch Senden eines voteReadOnly dem Koordinator signalisieren, dass keine Daten manipuliert wurden und somit eine Teilnahme am 2PC nicht notwendig ist. Des Weiteren wird bei nur einem Teilnehmer zum schnelleren Abschluss der Transaktion der One-Phase Commit angeboten (siehe BTP).

Das synchronization-Protokoll ist vergleichbar mit dem Volatile 2PC in WS-AT und erlaubt es registrierten Teilnehmern, vor dem Beginn und nach Beendigung des 2PC informiert zu werden (Nachrichten beforeCompletion und afterCompletion). In beiden Fällen wird dem Teilnehmer auch der Ausgang des 2PC mitgeteilt; im ersten Fall der voraussichtliche und im zweiten Fall der tatsächliche. Die beforeCompletion-Nachricht muss vom(von den) Teilnehmer(n) mit success bestätigt werden. Ist dies nicht der Fall beziehungsweise tritt in dieser Phase bei einem Teilnehmer ein Fehler auf, führt dies zu einem Abbruch beim Koordinator (wenn nicht sowieso schon vom Koordinator auf Rollback entschieden wurde). Ein Fehler nach Erhalt der afterCompletion-Nachricht hat hingegen keinerlei Auswirkungen auf die Transaktion. Abbildung 3.14 zeigt den Ablauf des synchronization-Protokolls.

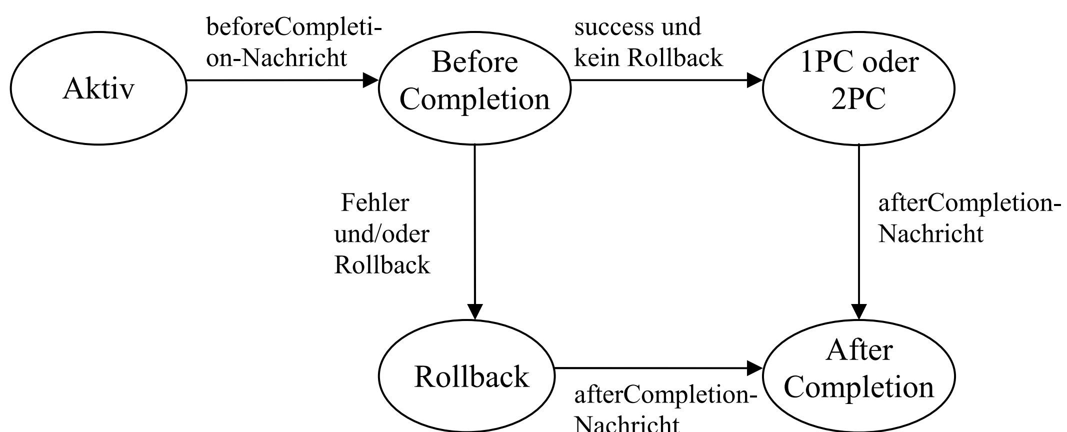


Abbildung 3.14: Ablauf des synchronization-Protokolls

Selbstverständlich ist beim ACID Transaktionsmodell auch Interposition möglich. Dies wird durch die Definition in WS-CF (siehe [Bunt03b]) erlaubt.

Auch bei AT steht es den Teilnehmern frei, sich direkt für einen Koordinator zu registrieren, oder dies über einen eigenen Koordinator zu tun.

Das Entfernen eines Teilnehmers von der Transaktion ist während deren gesamter Dauer nicht erlaubt.

Auf Recovery, das im Rahmen von AT auch behandelt wird und das in [Bunt03b] definierte Mechanismen verwendet, wird in dieser Arbeit nicht näher eingegangen. Auch die WSDL-Definitionen der Nachrichten werden hier aus Platzgründen nicht behandelt. Die interessierte LeserIn wird daher in beiden Fällen auf die Literatur verwiesen.

3.4.3 Long Running Action

Das Transaktionsprotokoll der Long Running Action (LRA, neu: WS-LRA) stellt ein Modell für lange Geschäftsinteraktionen, oder anders gesagt für die Verarbeitung langer Transaktionen dar. Eine LRA ist eine Aktivität, die aus einer oder mehreren Arbeiten besteht und eine Geschäftsinteraktion darstellt.

Um eine ausreichende Unterstützung von langen Transaktionen zu ermöglichen, ist es – wie bereits behandelt – notwendig, die ACID-Kriterien zu lockern und/oder andere Verhaltensweisen im Vergleich zu traditionellen Transaktionsformen zu definieren. Das Modell der Long Running Action bedient sich im Wesentlichen der folgenden Punkte:

- Isolation und Durabilität für LRAs sind in [Bunt03c] nicht festgelegt, sondern der Implementierung, also den Teilnehmern überlassen
- Arbeiten müssen kompensierbar sein
- Wie die Arbeiten und Kompensationen durchgeführt werden, ist Entscheidung der einzelnen Teilnehmer
- Interposition ist durch das Layering von WS-TXM af WS-CF möglich
- Jederzeitiges deregistrieren eines Teilnehmers einer Aktivität möglich
- Qualifier (z.B. Timelimit, das angibt, wie Lange ein Teilnehmer – genau genommen der Kompensator; siehe unten – die Kompensation einer Arbeit garantieren kann)

Da die Isolation von einander konkurrierenden Aktivitäten sowie die Durabilität nicht Teil des Protokolls sind, muss ein an der LRA teilnehmendes Service im Falle einer möglichen notwendigen Kompensation einen so genannten Kompensator ein-

setzen. Dieser Kompensator ist Teilnehmer an der LRA und arbeitet im Auftrag des Service.

Ist die gesamte Aktivität erfolgreich (Success), muss der Kompensator lediglich e-ventuell notwendige „Aufräumungsarbeiten“ durchführen, um die Konsistenz sicherzustellen. Ist die Aktivität nicht erfolgreich (Fail), muss er die Arbeit des Service wieder kompensieren. Dies kann beispielsweise durch Rückgängigmachung oder das Durchführen einer weiteren LRA erfolgen. Zweiteres bedeutet wiederum, dass diese kompensierende LRA wiederum einen Kompensator hat. Ist die Kompensation nicht möglich (cannotCompensateFault), muss der Kompensator – sofern das Problem nicht innerhalb der Applikation gelöst werden kann – alle relevanten Daten mitloggen, um eine Kompensation außerhalb der LRA zu ermöglichen. Die die LRA(s) steuernde Applikation kann von vorne herein festlegen, ob nur Teilnehmer erlaubt sind, die kompensieren können oder auch andere. Dieses geschieht mittels des MustUnderstand Attributs. Das Sequenzdiagramm in Abbildung 3.15 zeigt den möglichen Ablauf einer LRA. Die gesamten Interaktionsmöglichkeiten zwischen Teilnehmer (Kompensator) und Koordinator (LRA) werden nicht näher erläutert.

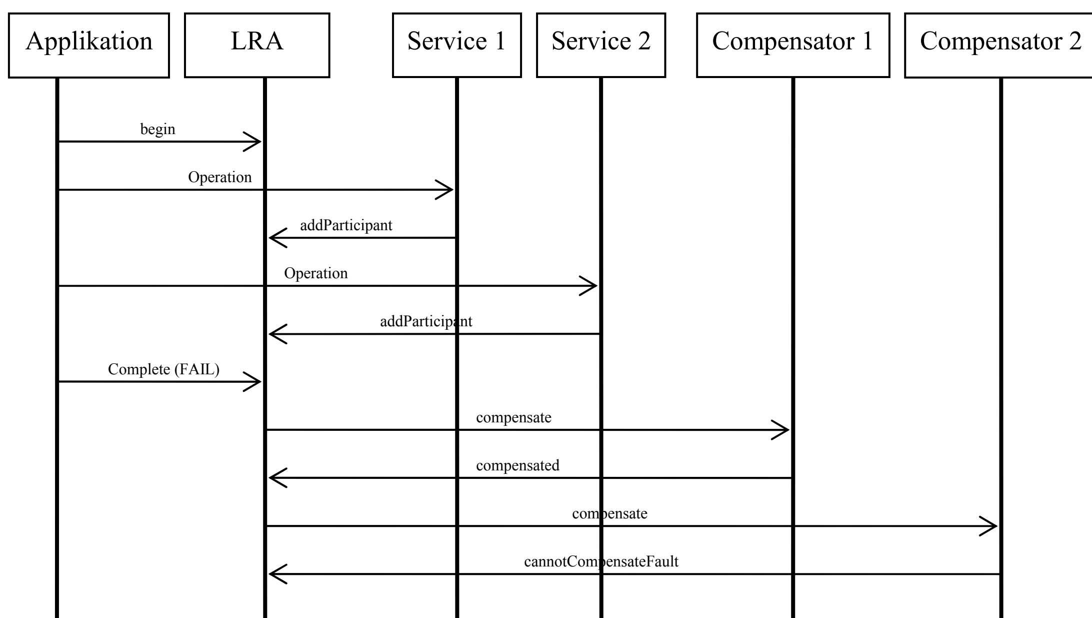


Abbildung 3.15: Beispielhafter Ablauf einer LRA

Verwendung von LRA

LRAs sind kompensierbare Arbeitspakete, die sowohl sequenziell als auch parallel miteinander kombiniert werden können. Jede LRA für sich ist zwar atomar, aber die

Kombination von „unabhängigen“ LRAs durch die Applikation erfordert dies nicht (sofern die LRAs natürlich nicht innerhalb einer LRA geschachtelt sind). Diese Umstände bieten der Applikation die Möglichkeit, Geschäftsprozesse entsprechend darzustellen und beispielsweise auf die Nichtdurchführbarkeit einer LRA mit der Durchführung einer anderen LRA zu reagieren (z.B. Buchung von Economy-Class Flug, wenn Business-Class nicht mehr verfügbar). Des Weiteren ist es möglich, mehrere LRAs auszuführen und wenn, wie bei den Cohesions beim BTP (siehe Unterkapitel 3.2) eine geeignete Untermenge von LRAs (z.B. geeignetes Package) vorhanden ist, diese komplett durchzuführen und alle anderen LRAs kompensierend zu beenden. Wie dies für das Holiday Package Beispiel aussehen kann, wird etwas später in diesem Unterkapitel behandelt.

3.4.4 Business process transaction

Die dritte Transaktionsform, die von WS-CAF zur Verfügung gestellt wird, ist das der Business process transaction (hier BPT abgekürzt, neu: WS-BP). Diese zielt darauf ab, den Ablauf und Ausgang von Geschäftsprozessen innerhalb eines Transaktionskontexts zu gewährleisten. Ein Geschäftsprozess (Business Process) besteht aus mehreren so genannten Business Tasks und jeder dieser Business Tasks wird innerhalb einer so genannten Business Domain ausgeführt. Jede Business Domain wird durch einen so genannten BusinessTaskCoordinator repräsentiert. Eine Business process transaction stellt nur einen Geschäftsprozess beziehungsweise eine Aktivität dar und ist für die Interaktion – die sowohl synchron als auch asynchron von BPT unterstützt wird – der einzelnen Business Domains verantwortlich.

Aufgrund der oft hohen Komplexität bei Geschäftsprozessen spielt Interposition bei BPT eine große Rolle. Eine Business Domain kann so selbst eine BPT sein und wiederum aus mehreren Sub-Domains bestehen und so weiter. Abbildung 3.16 veranschaulicht eine mögliche Strukturierung.

Der Abschluss eines Geschäftsprozesses ist atomar – entweder alle angefragten Arbeiten werden erfolgreich abgeschlossen (confirm), oder alle werden rückgängig gemacht (cancel). Business process transaction verfolgt bei der Ausführung von Tasks ein optimistisches Modell. Das bedeutet, dass davon ausgegangen wird, dass der Fehlerfall eine Ausnahme darstellt und der Ablauf im Regelfall fehlerfrei ist. Tritt dennoch der Fehlerfall ein, wird davon ausgegangen, dass alle Tasks kompensierbar sind. Wie die Kompensation - beziehungsweise auch der Task im Normalfall – durchgeführt wird, ist nicht Teil der Spezifikation. Dies erfordert Logging und

unter Umständen eine Kompensation außerhalb der Transaktion (z.B. muss der Administrator informiert werden, der die Arbeit dann „händisch“ kompensiert).

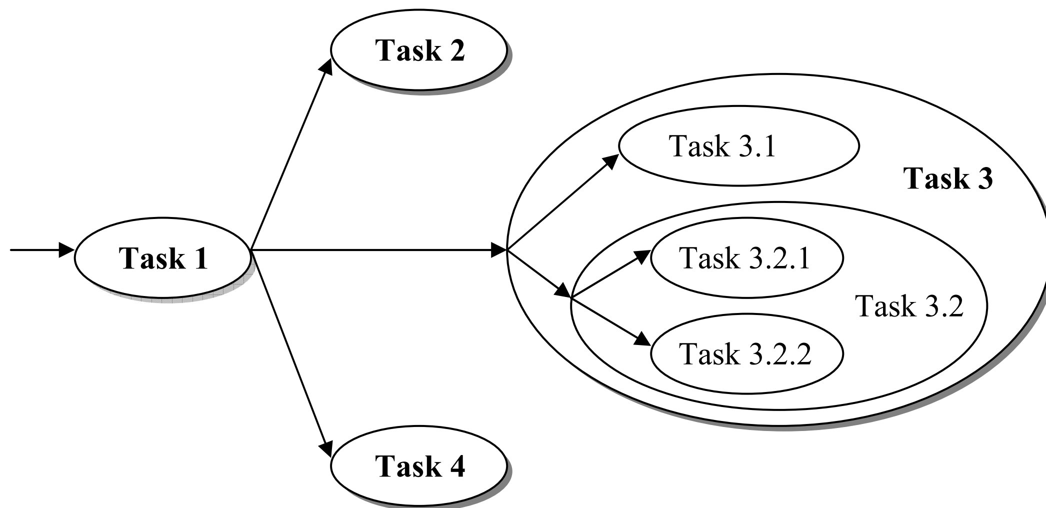


Abbildung 3.16: Business Processes und Tasks

Der Begriff Kompensation an sich unterscheidet sich beim BP Modell von der herkömmlichen Definition. Kompensation bedeutet nicht zwangsläufig eine Rückgängigmachung oder Aufhebung von Inkonsistenzen. Tritt beispielsweise ein Fehler innerhalb einer Business Domain auf, so kann Kompensation im Sinne von Geschäftsprozessen auch derart aussehen, dass der komplette Task vom Prozess entfernt wird, oder aber andere Tasks anstelle des geplanten ausgeführt werden (beispielsweise wenn der Task „Flug buchen“ fehlschlägt, könnte die Kompensation das Buchen eines Zugtickets bedeuten (natürlich neben der „normalen“ Kompensation der fehlgeschlagenen Flugbuchung)).

Es kann also durchaus sein, dass die Transaktion vom Fehlerstatus wieder in den Arbeitsstatus zurückkehrt. Abbildung 3.17 (aus [Bunt03c]) zeigt die einzelnen Statuswechsel-Möglichkeiten.

Ein weiterer dehnbarer Begriff beim BP Modell ist der der Business Domain. Wie eine Business Domain „im Inneren“ aussieht beziehungsweise wie sie implementiert ist, wird in [Bunt03c] freigestellt. Einzige Anforderung ist das Verstehen der BPT Protokoll-Nachrichten. Eine Business Domain kann also beispielsweise einen BTP-Koordinator (siehe Unterkapitel 3.2) haben, der das Mapping der BPT Protokoll-Nachrichten für BTP Atoms und Cohesions übernimmt. Eine Domain kann aber ge-

nauso gut eine ACID transaction oder eine LRA sein. Solange die Teilnehmer das BPT Protokoll befolgen, kann die Implementierung nicht festgestellt werden ([Bunt03c]).

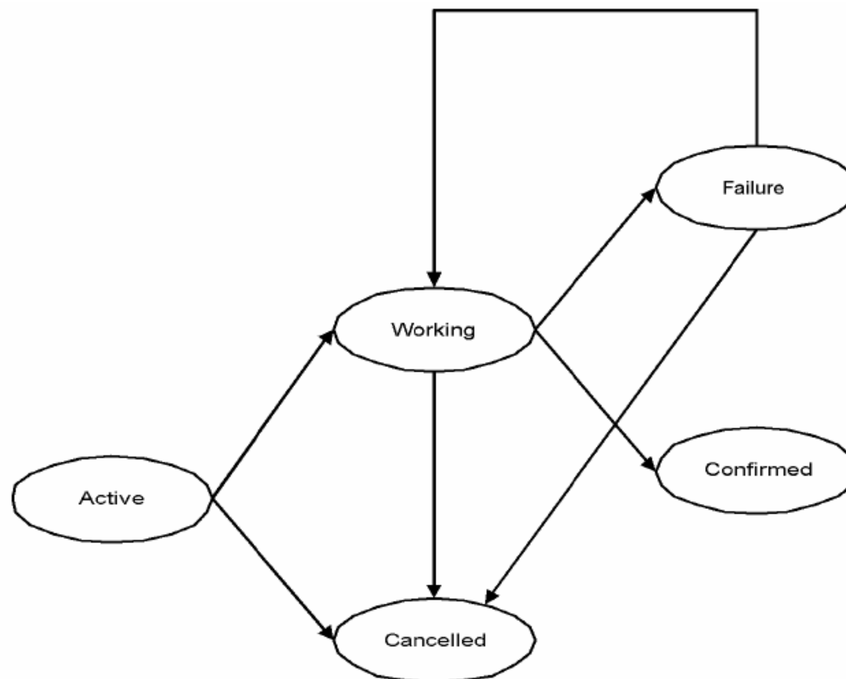


Abbildung 3.17: Statuswechsel eines Business Processes (aus [Bunt03c])

Protokolle

Die Einteilung der in BPT verwendeten Protokolle zur Steuerung der Transaktion erfolgt von Bunting et al. (in [Bunt03c]) in zwei Kategorien:

- getrieben von der Business Domain (driven from the Business Domain)
- getrieben zur Business Domain (driven to the Business Domain)

Aus Gründen des Umfangs werden die Protokolle nur kurz vorgestellt. Genaue Erklärungen zu Nachrichten und WSDL-Dokumenten können in [Bunt03c] nachgelesen werden.

Von der Business Domain:

- terminate-notification Sub-Protokoll
  URI:
  `http://www.webservicestransactions.org/wstxm/tx-bp/tn/2003/03`
  Mittels dieses Protokolls teilt der Koordinator (hier CoordinatorParticipant) dem(den) Teilnehmer(n) am Protokoll (TerminatorParticipant) automatisch mit, ob der Geschäftsprozess erfolgreich beendet werden kann oder nicht.

Abbildung 3.18 (aus [Bunt03c]) zeigt die möglichen Nachrichten.

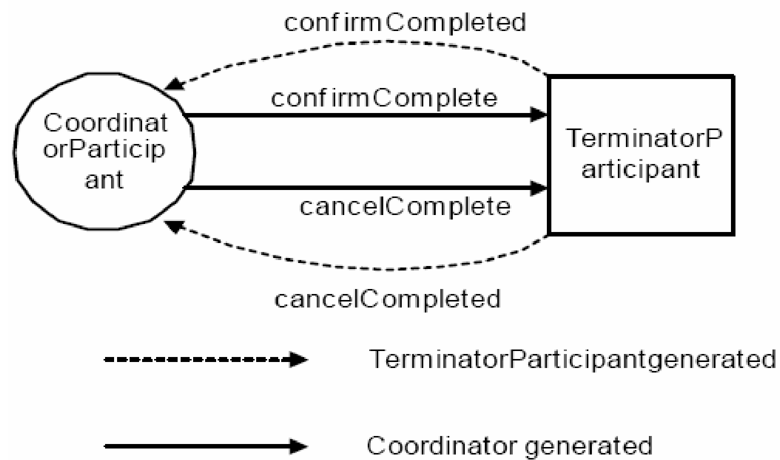


Abbildung 3.18: terminate-notification Protokoll (aus [Bunt03c])

- businessProcess Protokoll
  URI:
  `http://www.webservicestransactions.org/wstxm/tx-bp/bp/2003/03`
  Dieses Protokoll dient dem Parent (hier BusinessProcessParticipant) einer Domain (hier CoordinatorParticipant), zu erfahren, wenn die angeforderte Arbeit nicht durchgeführt werden konnte (failure) beziehungsweise bei Nichtdurchführbarkeit zu erfahren, wenn das komplette Canceln nicht möglich war (failureHazard). Die Informationen geben dem Parent die Möglichkeit, Kompensationen durchzuführen. Abbildung 3.19 (aus [Bunt03c]) zeigt wieder den Nachrichtenfluss.

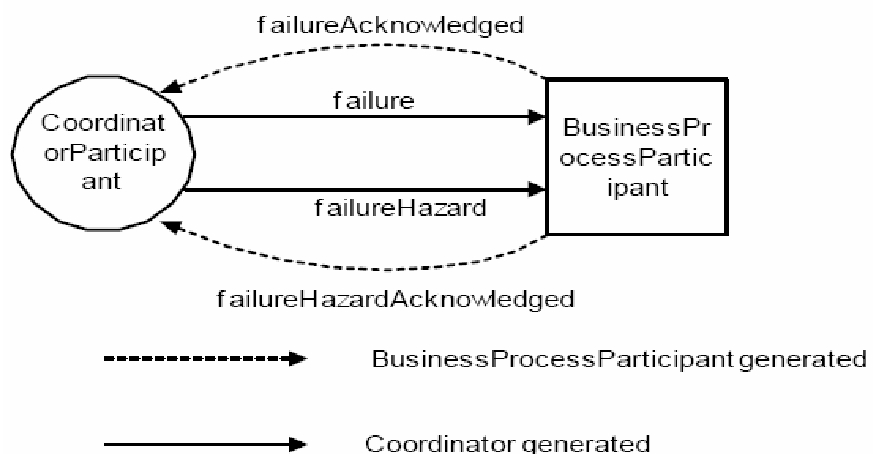


Abbildung 3.19: businessProcess Protokoll (aus WS-TXM)

Zur Business Domain:

Bei diesen Protokollen gibt es drei Teilnehmergruppen: den Client (die Applikation), den Koordinator der Business Process Transaction und die Business Domain. Der Nachrichtenfluss erfolgt derart, dass vom Client Anfragen an den Koordinator gestellt werden, dieser eine entsprechende Anfrage an die Business Domain (genau: den BusinessTaskCoordinator) stellt und die zugehörigen Antwortnachrichten in umgekehrter Reihenfolge wieder „zurückfließen“.

- checkpoint Protokoll
  URI:
  `http://www.webservicestransactions.org/wstxm/tx-bp/cp/2003/03`
  Dieses wird zur Erzeugung von Wiederherstellungspunkten verwendet. Business Domains erstellen bei Aufforderung Haltepunkte (checkpoints), von denen später wieder gestartet werden kann. Durch Angabe des checkpointTimelimit Qualifiers kann eine Domain mitteilen, wie lange sie bereit ist, zu versuchen, den checkpoint zu speichern. Jeder checkpoint besitzt eine eigene ID. Die Ausführung des Protokolls kann so oft wie gewünscht erfolgen.
  Abbildung 3.20 (aus [Bunt03c]) zeigt das Protokoll wieder grafisch. Die angegebene Legende wird aus Platzgründen in den nachfolgenden Abbildungen ausgespart.

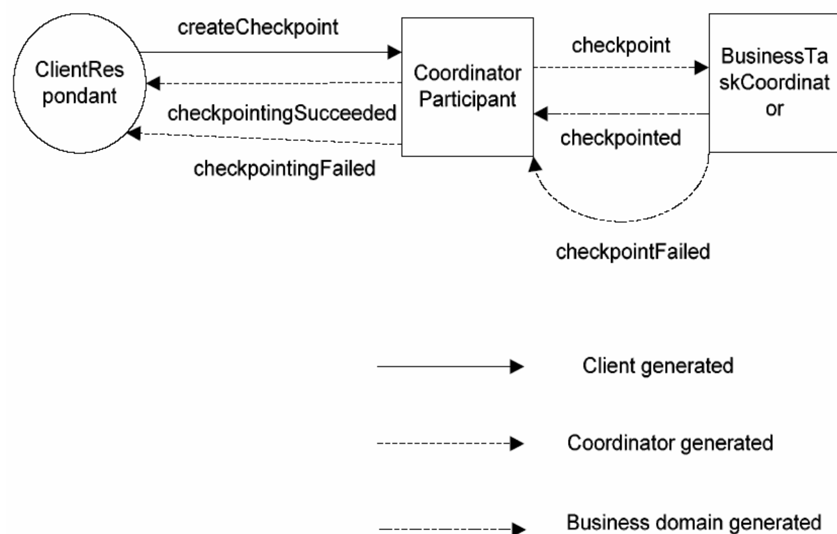


Abbildung 3.20: checkpoint Protokoll (aus [Bunt03c])

- restart Protokoll
  URI:

`http://www.webservicestransactions.org/wstxm/tx-bp/restart/2003/03`

Dient zum restart und veranlasst bei Ausführung die Domains zum restart vom angegebenen checkpoint. Das restart Protokoll kann so oft wie gewünscht ausgeführt werden.

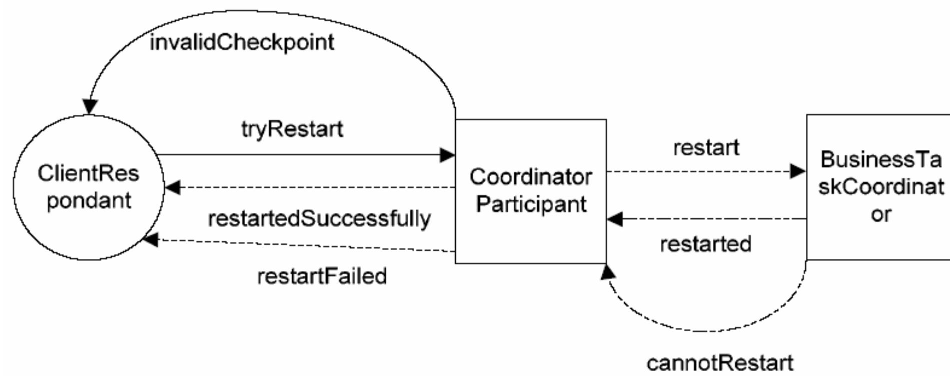


Abbildung 3.21: restart Protokoll (aus [Bunt03c])

- workStatus Protokoll
  URI:
  `http://www.webservicestransactions.org/wstxm/tx-bp/ws/2003/03`
  Mittels dieses Protokolls kann der Client den Status der Domain(s) abfragen (ob sie bereits beendet beziehungsweise abgebrochen haben oder noch bei der Verarbeitung sind). Die Ausführung des Protokolls kann beliebig oft erfolgen.

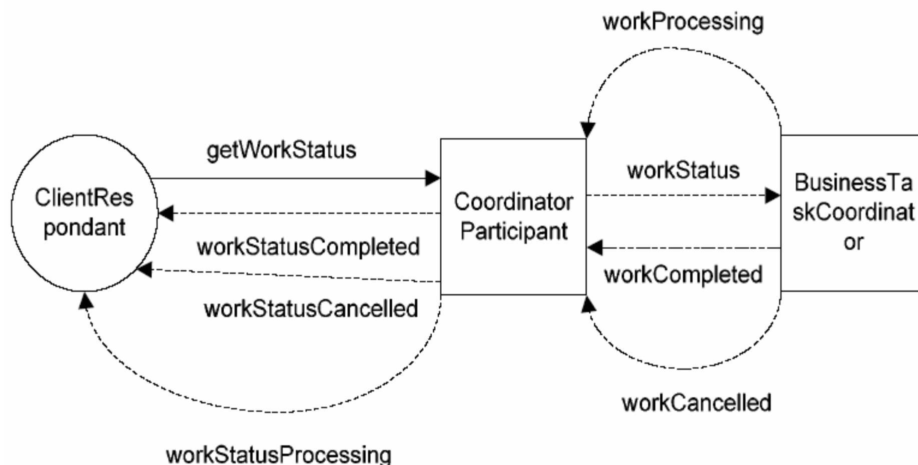


Abbildung 3.22: workStatus Protokoll (aus [Bunt03c])

- completion Protokoll
  URI:
  `http://www.webservicestransactions.org/wstxm/tx-`

`bp/completion/2003/03`

Das Protokoll dient zur Beendigung aller Business Domains. Das completion Protokoll kann nur einmal ausgeführt werden, wenn die WS-CTX Aktivität terminiert ([Bunt03c]). Die Transaktion wird also am Ende des Geschäftsprozesses beendet. Abbildung 3.23 zeigt die möglichen Interaktionen und wurde [OASI05] entnommen.

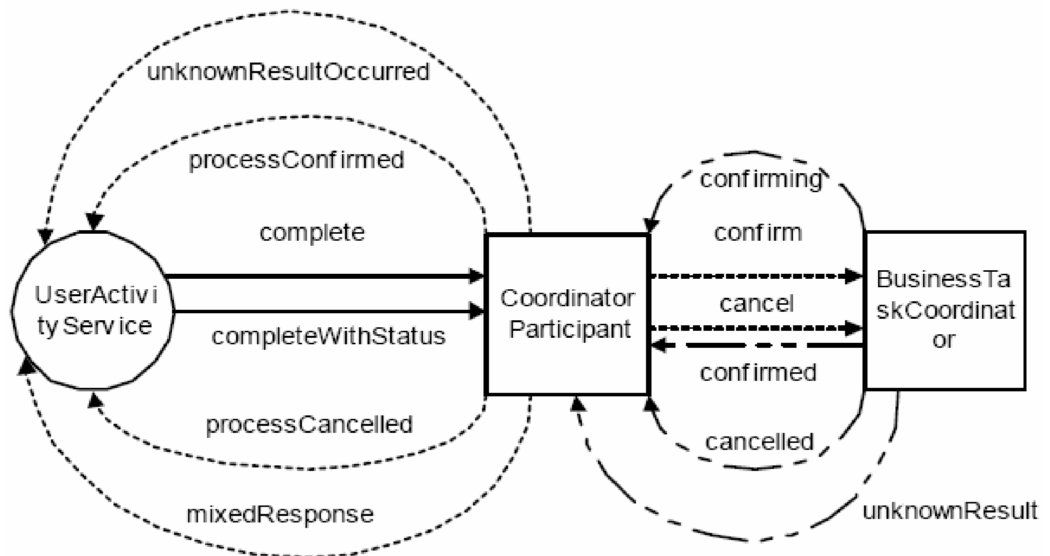


Abbildung 3.23: completion Protokoll (aus [OASI05])

3.4.5 WS-CAF und Holiday Package

Mit den drei gebotenen Transaktionsformen AT, LRA und BPT ist eine gute Möglichkeit der Adaptierung an individuelle Aufgabenstellungen von Seiten des Web Services Composite Application Frameworks gegeben.

Während ACID transaction für die Integration bestehender ACID Systeme und das Sicherstellen der vier ACID Kriterien bei Bedarf relevant sind und darüber hinaus mit der Möglichkeit der Interposition erlaubt sowie durch das synchronization Protokoll Caching unterstützt, stellen LRA und BTP weitaus interessantere Instrumente für das Holiday Package dar, wenngleich wiederum betont werden muss, dass ACID Transaktionen vor allem im Finanzbereich nach wie vor von größter Relevanz sind.

Die bereits bei den LRAs kurz angesprochene Möglichkeit der Kombination von atomaren LRAs und deren Steuerung über die Applikation lässt eine sehr gute Modellierung des Holiday Packages zu, wie Abbildung 3.24 zeigt.

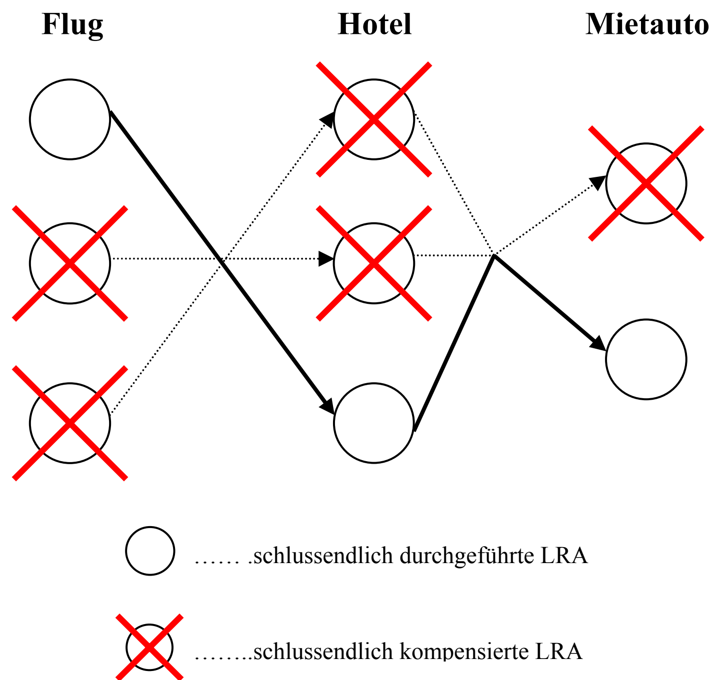


Abbildung 3.24: Kombination von LRAs

Jede einzelne in Abbildung 3.24 eingezeichnete LRA könnte beispielsweise – durch die von LRA unterstützte Interpositionsmöglichkeit – aus den Operationen Preisabfrage und Bestellung bestehen. Zuerst wird jeweils der Preis abgefragt und schlussendlich nur jene LRAs komplett ausgeführt (Buchung), deren Preis am niedrigsten ist, alle anderen werden mittels Kompensation wieder ungeschehen gemacht. Zu beachten ist, dass alle LRAs in Abbildung 3.24 eigentlich unabhängig voneinander sind und die Applikation je nach Bedarf LRAs ausführt beziehungsweise kompensiert. Die Buchung des Mietautos könnte beispielsweise so ablaufen, dass zuerst nur bei einem Anbieter angefragt wird, allerdings bei auftretenden Problemen die LRA kompensiert wird und anstatt dessen von der Applikation mit der Ausführung einer alternativen Buchung darauf reagiert wird (siehe Abbildung 3.24). Die Preisabfragen für die Flugbuchung und das Hotel könnten jeweils parallel erfolgen.

LRAs bieten durch die Lockerung des Isolations-Kriteriums (Entscheidung der Implementierung) eine gewisse Freiheit für die Teilnehmer, einen eher optimistischen oder pessimistischen Ansatz zu wählen und somit individuelle Gegebenheiten optimieren zu können (Stichwort Nebenläufigkeit). Die durch die Lockerung der Isolation notwendige Kompensation ist den Teilnehmern in ihrer Art und Weise zwar freigestellt, doch erfordert diese das oft lange Speichern von für die Kompensation notwendige Informationen, wenngleich die zeitliche Unbekannte durch setzen eines Timelimits vom Kompensator (wie lange kann kompensiert werden) konkretisiert

werden kann und dadurch das rechtzeitige Setzen entsprechender Handlungen von Seiten der Applikation möglich wird. Für das Holiday Package bedeutet die Freiheit bei der Kompensation und das Setzen von Zeitlimits weitere Anpassungs- und Optimierungsmöglichkeiten an verschiedenste Problemstellungen.

Durch die Ausrichtung auf Geschäftsprozesse ist die Transaktionsform Business process transaction geradezu prädestiniert für das Holiday Package Beispiel. Die Aufteilung der Arbeit in mehrere Tasks, die synchrone beziehungsweise asynchrone Kommunikation und die Interposition bieten die Möglichkeit der Abbildung sehr komplexer Geschäftsprozesse. Dabei ist durch die individuellen Gestaltungsmöglichkeiten der Business Domains der gleichzeitige Einsatz verschiedener WS-TXM Transaktionsformen (AT, LRA, BPT) sowie der Einsatz verschiedener Transaktionsprotokolle (BTP, WS-C und WS-Tx, …) zum Einen eine ideale Anpassung an Bedürfnisse von Geschäftsprozessen und zum Anderen die Integration bestehender Komponenten sowie die in manchen Fällen geeignetere Verwendung von „Konkurrenz"-Protokollen und Systemen durch ummappen auf BPT möglich. Zur Abwicklung einer Transaktion mit heterogenen Systemen wurde damit eine gute Basis geschaffen.

Des Weiteren positiv bei BPT ist die Kompensation. Diese erfordert zwar auch logging, doch durch ihre weitere Bedeutung als lediglich Rückgängigmachen oder Wiederherstellen kann auf Interaktionen mit Anwendern und daraus resultierende Handlungen dynamisch reagiert werden.

Auch wird eine Vielzahl an Protokollen zur Kommunikation unter den Teilnehmern zur Verfügung gestellt. Die im Zusammenhang mit dem Holiday Package hervorzuhebenden sind dabei das businessProcess Protokoll, das den Teilnehmern ermöglicht, auftretende Fehler mitzuteilen und der Applikation dadurch die Chance gibt, unmittelbar darauf zu reagieren, ohne das Ende der Transaktion abwarten zu müssen, sowie das completion Protokoll, das – im Gegensatz zu den WS-BusinessActivities (vgl. Unterkapitel 3.3) – die Beendigung einer Transaktion spezifiziert – was allerdings aufgrund der etwas anderen Architektur vielleicht etwas einfacher ist. Auch das workStatus- und das checkpoint-Protokoll können sich bei der Anwendung insofern als nützlich erweisen, als dass durch Statusabfragen stets aktuelle Informationen über Domains herangezogen werden können und durch die Checkpoints auf bereits erreichte Zwischenergebnisse zurückgegriffen werden kann.

Alles in Allem ist WS-CAF sehr umfassend und für verschiedenste Transaktionsarten und damit auch für das Holiday Package geeignet. Durch das Layering auf WS-

CTX und WS-CF sind Erweiterungen und Änderungen ohne große Architekturänderungen möglich. Weiters ist durch die Auslegung des gesamten Frameworks auf Web Services auch die nötige Kompatibilität mit anderen Systemen gewährleistet.

Bleibt nur abzuwarten, wie derzeit in Gang befindlichen Änderungen der in drei Teile aufgesplitteten WS-TXM Spezifikation aussehen werden und inwieweit die Transaktionsformen ihre Funktionalität erhalten/erweitern/ändern. Möchte man WS-CAF zum heutigen Zeitpunkt verwenden, muss man sich die Frage stellen, ob es Sinn macht, eine Version zu verwenden, die gerade einer Änderung unterzogen wird, oder ob man nicht warten soll, bis die neue Version verabschiedet wird (was aus heutiger Sicht allerdings noch nicht absehbar ist). Schließlich werden Neuerungen im Regelfall nur aus Bedarfsgründen und/oder zur Weiterentwicklung gemacht.

3.5 Gegenüberstellung der Protokolle

Nachdem die Protokolle BTP, WS-C/WS-Tx und WS-CAF einzeln beleuchtet wurden und jeweils ihre Eignung für das Holiday Package Beispiel diskutiert wurde, folgt nun eine grobe Gegenüberstellung der Protokolle anhand von zehn für das Holiday Package relevanten Kriterien. Die Kriterien werden nun samt einer Begründung für deren Relevanz kurz vorgestellt.

Da das Holiday Package (genaue Beschreibung siehe Unterkapitel 1.1) aus mehreren verschiedenen Teilen besteht (Flug, Hotel, Mietauto), deren Wichtigkeit für das Gesamtpaket auch unterschiedlich ist/sein kann – die Buchung eines Mietautos ist beispielsweise eher sekundär, wichtig sind in erster Linie Flug und Hotel – stellen die Kriterien **Interpositionsmöglichkeit** und **Schachtelung** sowie die Unterscheidung zwischen **vitalen und nicht vitalen Teilen** wichtige Eckpfeiler für die Durchführung des holiday Packages dar. Weiters ist auch die **Konzipierung** eines Protokolls speziell **für Web Services** im Hinblick auf die Interoperabilität zwischen verschiedenen (heterogenen) Systemen von größter Relevanz.

Das Holiday Package zwar weniger betreffend ist auch die Frage, ob mit den Protokollen Transaktionen, die den ACID-Kriterien genügen müssen möglich sind, eine wichtige, da **ACID** für viele Problemstellungen interessant und notwendig ist.

Neben den Kriterien „**leichte Erweiterbarkeit**“ der Protokolle und der Möglichkeit, **mehrere Protokolle kombinieren** zu können, um so bestmögliche Anpassungen an individuelle beziehungsweise neue Probleme zu ermöglichen, ist im Bereich der Verarbeitung von langen Transaktionen, wie sie im Holiday Package benötigt wer-

den, das Verhalten der Teilnehmer im Hinblick auf die Auswirkung auf die Nebenläufigkeit (Stichwort Performanz) äußerst wichtig. Die Problem der Erhöhung Nebenläufigkeit kann durch vorzeitiges Publikmachen von (Teil-)Ergebnissen in Kombination mit Kompensation adäquat gelöst werden (Kriterium **„vorzeitiges Buchen; Kompensation“**). Weiters ist auch eine **Freistellung der Entscheidung** für jeden Teilnehmer an der Transaktion über die Art und Weise seines Verhaltens (z.B. kann Teilnehmer selbst entscheiden, ob er blocken oder bereits provisorisch buchen möchte?) zur Optimierung der Performanz beziehungsweise der Anpassung an individuelle Gegebenheiten im Umfeld eines Teilnehmers ein wichtiges Kriterium.

Letztes der zehn Kriterien ist die Frage nach den Möglichkeiten, die die Protokolle bieten, wenn Fehler auftreten (z.B. Flug kann doch nicht gebucht werden). Muss der gesamte Prozess abgebrochen werden – sprich schlägt die Transaktion fehl - , oder können bereits erledigte Teile erhalten bleiben und ist nur eine neuerliche beziehungsweise alternative Ausführung der fehlerhaften Teile nötig? (Kriterium „**Fortsetzen der Transaktion bei auftretenden Fehlern**“)

Zur Bewertung der vorgestellten Kriterien werden drei Symbole verwendet:

+ …. Kriterium wird vom Protokoll erfüllt
- …. Kriterium wird vom Protokoll nicht erfüllt
~ …. Erfüllung des Kriteriums ist teilweise oder mit Vorbehalten möglich

Kriterium	**BTP**	**WS-C/WS-Tx**	**WS-CAF**
Interposition	+	+	+
Schachtelung	-	+	+
Vitale/nicht vitale Teile[5]	~	~	~
Konzipierung für Web Services	-	+	+
ACID Transaktionen möglich	-	+	+
Leichte Erweiterung	~	+	+
Kombinieren verschiedener Pro-	~	~	+

[5] Bei allen drei Protokollen nicht explizit möglich, aber logisch/mit Applikationslogik lösbar

tokollen			
Vorzeitiges Buchen; Kompensation	+	+	+
Freistellung der Entscheidung	+	-	+
Fortsetzen der Transaktion bei auftretenden Fehlern	-	+	+

Tabelle 3.3: Gegenüberstellung der Protokolle

Die Gegenüberstellung in Tabelle 3.3 zeigt grob, dass WS-CAF und WS-C/WS-Tx gemessen an den ausgewählten Kriterien sehr gut für die gegebene Problemstellung sind, wohingegen das BTP einige Schwächen aufweist. Vor allem WS-CAF bietet sehr viele verschiedene Möglichkeiten, Transaktionen durchzuführen und stellt damit ein ideales Framework für die Durchführung des Holiday Package Beispiels oder ähnlich komplexer Aufgabenstellungen dar. Auch WS-C/WS-Tx weist durchaus ideale Eigenschaften für derartige Problemstellungen auf.

Was bei den Kriterien nicht berücksichtigt wurde, aber dennoch von Relevanz sein kann, ist die Tatsache, dass mehr Möglichkeiten eine höhere Komplexität des Protokolls beziehungsweise der Spezifikation bedeuten kann. So empfand ich etwa die Spezifikation von WS-C/WS-Tx wesentlich kompakter und damit leichter verständlich und übersichtlicher, als jene von BTP und WS-CAF. Dieser Eindruck beruht selbstverständlich auf rein subjektiver Wahrnehmung und kann daher nicht als Vergleichskriterium herangezogen werden.

Kapitel 4

Transaktionen in J2EE und .Net

Nachdem nun die Transaktionsmöglichkeiten bei Web Services untersucht wurden, beschäftigt sich dieses Kapitel mit der Durchführung und den Möglichkeiten von Transaktionen in J2EE und .Net. Dabei stehen vor allem die für den Anwender gebotenen Möglichkeiten und die praktische Handhabung im Vordergrund, ohne zu sehr auf die theoretischen Konzepte einzugehen. Des Weiteren wird die Eignung im Hinblick auf das Holiday Package in gewohnter Manier eruiert.

4.1 Transaktionen in J2EE

Transaktionen sind ein zentraler Punkt in J2EE und werden in der für diese Arbeit herangezogene Spezifikation [Sun03] explizit behandelt.

Bei Transaktionen in J2EE spielen im Wesentlichen vier Komponenten zusammen:

- Applikation
- Applikationsserver
- Ressourcemanager
- Transaktionsmanager

Wie in Abbildung 4.1 dargestellt, sieht die Zusammenarbeit zwischen den Komponenten so aus, dass die Applikation, die am Applikationsserver läuft, Ressourcen verwendet, die vom(n) Ressourcemanager(n) verwaltet werden. Ressourcen können beispielsweise Datenbanken sein, die von einem RDBMS (Relational Database Ma-

nagement System) wie etwa Oracle koordiniert werden. Oracle 10g ist beispielsweise auch ein Ressourcemanager. Sollen mehrere Ressourcen in einem transaktionalen Kontext verwendet werden, ist es notwendig, diese untereinander zu koordinieren und unter ihnen einen konsistenten Ausgang herbeizuführen. Sind nur Ressourcen eines Ressourcemanagers betroffen, startet dieser selbst die Transaktion und übernimmt selbst die Koordination. Man spricht in diesem Fall von einer lokalen Transaktion. Nehmen Ressourcen mehrerer Ressourcemanager an einer Transaktion teil, so nennt man diese eine globale Transaktion und man benötigt einen Transaktionsmanager. Die Aufgabe des Transaktionsmanagers ist es, die Koordination von globalen Transaktionen zu übernehmen, also einen konsistenten Ausgang unter den Ressourcemanagern herbeizuführen.

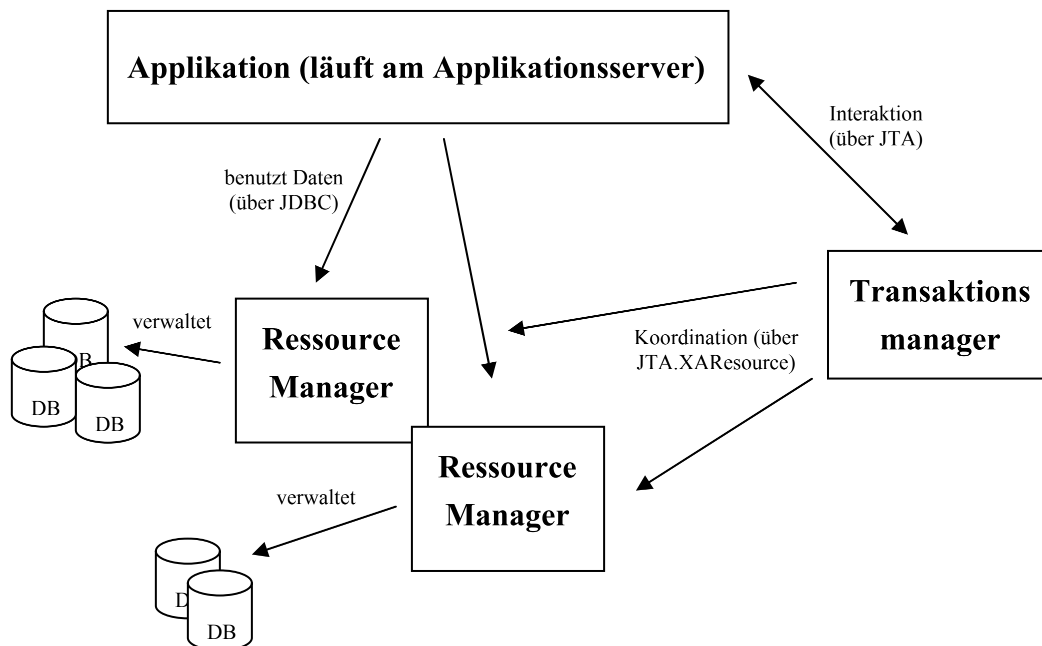


Abbildung 4.1: Zusammenspiel der Komponenten

Der Transaktionsmanager selbst kann als eigener Prozess laufen, oder aber im Applikationsserver integriert sein. Zur Kommunikation mit Applikation, Applikationsserver und Ressourcemanager muss der Transaktionsmanager Java-Interfaces zur Verfügung stellen. Diese sind in der Java Transaction API (JTA, [Sun02]) zusammengefasst. Die einzelnen Schnittstellen sind `javax.transaction.UserTransaction` für die Applikation, `javax.transaction.TransactionManager` für den Applikationsserver sowie `javax.transaction.xa.XAResource` für den(die) Ressourcemanager.

Der vom Transaktionsmanager angebotene Transaktionsservice, der JTA unterstützt, kann zwar grundsätzlich frei gewählt werden, doch wird von JTA das Java Transaction Service (JTS, [Sun99]) als zugrunde liegendes Service empfohlen. Welchen Transaktionsservice der Transaktionsmanager verwendet, ist vor allem im Hinblick auf die Kommunikation mit anderen Applikationsservern beziehungsweise Transaktionalen Systemen wichtig. JTS ist ebenso wie JTA eine Spezifikation von Sun Microsystems und definiert sich wie folgt:

> *"JTS specifies the implementation of a transaction manager which supports the JTA specification at the high-level and implements the Java mapping of the OMG Object Transaction Service (OTS) 1.1 specification at the low level."* [Sun99*]*

OTS ([OMG01]) ist ein Standard der OMG, der die Transaktionssteuerung regelt. In Abbildung 4.2 (aus [Sun99]) wird der Zusammenhang der erläuterten Teile grafisch veranschaulicht.

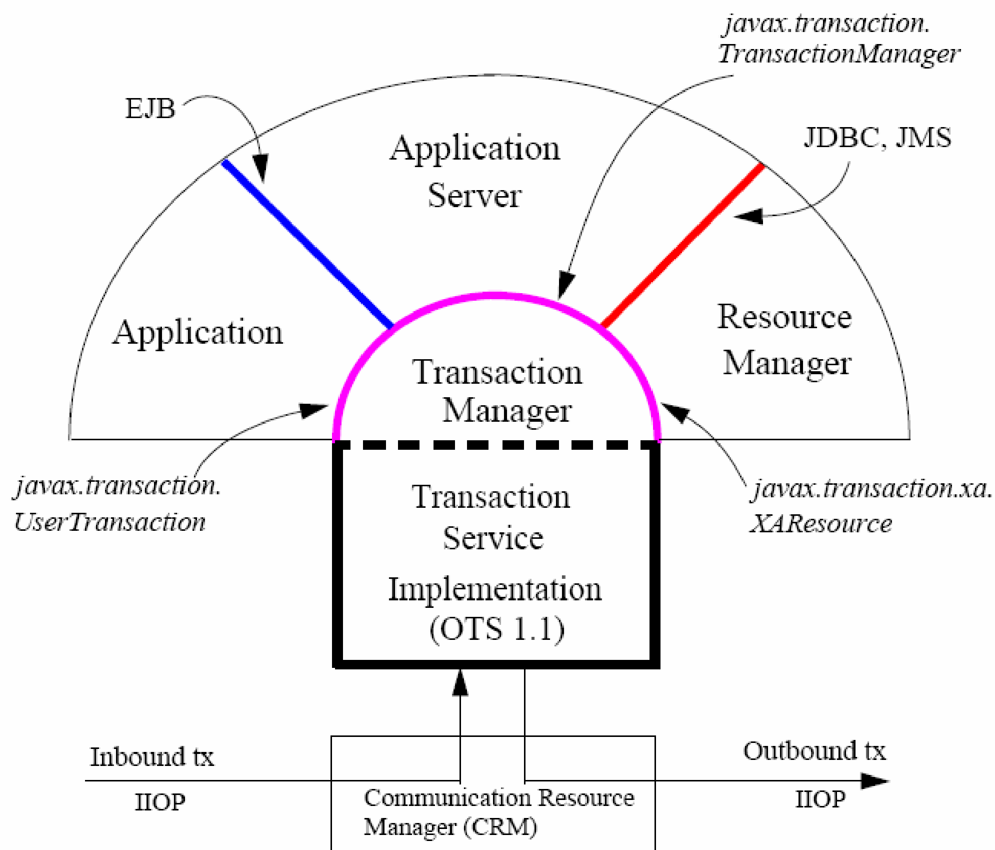


Abbildung 4.2: Zusammenhang der Komponenten (aus [Sun99])

Der in Abbildung 4.2 eingezeichnete Communication Resource Manager (CRM) ermöglicht die Interoperabilität zwischen Transaktionsmanagern, wird in JTA und JTS nicht definiert und obliegt dem Applikationsserver. Aus diesen Gründen wird diese Komponente hier nicht näher behandelt.

Objekttypen in JTS und OTS

Von OTS und JTS werden im Rahmen von Transaktionen drei Objekttypen anhand ihrer Rolle in einer Transaktion unterschieden, nämlich transactional clients (transaktionale Clients), transactional objects (transaktionale Objekte) und recoverable objects (wiederherstellbare Objekte), die sich allesamt auf das EJB-Konzept mit Web Komponenten (JSP, Servlet), EJBs (Session- und Entity Beans) und dahinterstehende Datenbanken umlegen lassen.

Die drei Objekttypen werden in [OMG01] folgendermaßen definiert (Auszug):

- Transactional clients: *A transactional client is an arbitrary program that can invoke operations of many transactional objects in a single transaction.*
- Transactional objects: *We use the term transactional object to refer to an object whose behavior is affected by being invoked within the scope of a transaction.*
- Recoverable objects: *An object whose data is affected by committing or rolling back a transaction is called a recoverable object.*

Transaktionale Clients können beispielsweise Servlets sein, die auf EJB Komponenten zugreifen und können die Steuerung einer Transaktion – z.B. starten und beenden – übernehmen. Neben dieser gibt es noch andere Möglichkeiten der Transaktionssteuerung, die in Abschnitt 4.1.1 näher erläutert werden.

Transaktionale Objekte speichern selbst keine Daten, nehmen aber insofern an der Transaktion teil, als sie diese zum Abbruch zwingen können (z.B. bei Auftreten von Exceptions). Jede EJB ist ein transaktionales Objekt.

Wiederherstellbare Objekte sind Objekte von den eigentlichen, persistent gespeicherten Daten (Ressourcen), wie etwa einzelne Rows einer Datenbanktabelle. Per Definition ist ein wiederherstellbares Objekt gleichzeitig ein transaktionales Objekt, umgekehrtes ist nicht der Fall.

Abbildung 4.3 aus [OMG01] stellt das Konzept mit den beteiligten Objekten noch einmal schematisch dar:

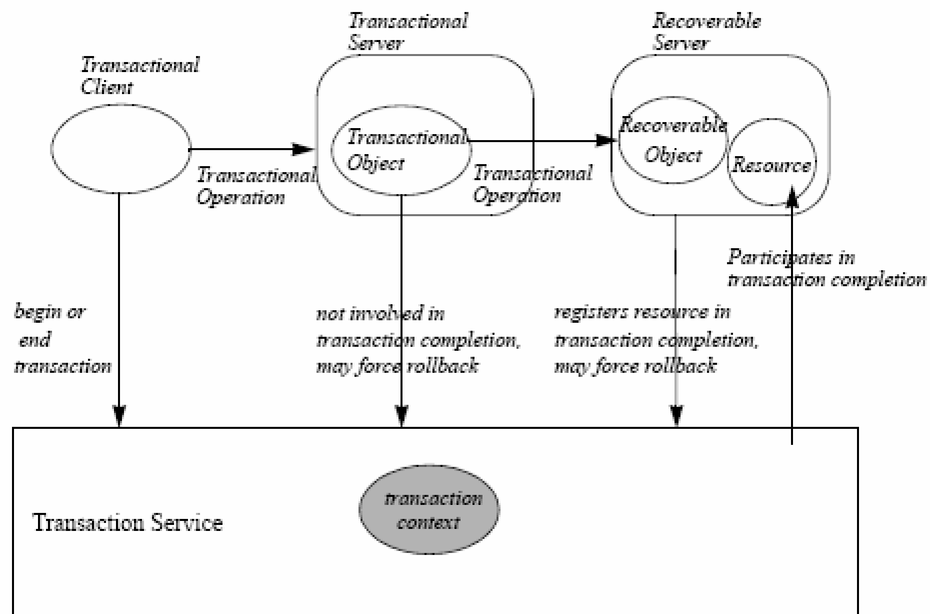


Abbildung 4.3: Interaktion der Objekttypen (aus [OMG01])

4.1.1 Transaktionssteuerung

Zur Steuerung von Transaktionen gibt es in J2EE grundsätzlich drei Möglichkeiten, die sich in zwei Gruppen einteilen lassen:

- Implizite Transaktionssteuerung (Container Managed Transactions)
- Explizite Transaktionssteuerung (Bean Managed Transactions und Transaktionssteuerung über Web Komponenten)

Implizite Transaktionssteuerung (Container Managed Transactions)

Wie der Name schon sagt, werden Container Managed Transactions vom EJB-Container verwaltet, sprich der EJB-Container übernimmt die Transaktionssteuerung. Container Managed Transactions können für jegliche Art von Enterprise Bean (Session Beans, Entity Beans, Message-driven Beans) verwendet werden und funktionieren so, dass der Container vor Aufruf einer Methode eine Transaktion erzeugt und diese vorm Verlassen der Methode (mit Commit) beendet.

Die Festlegung transaktionalen Verhaltens einer Methode erfolgt nicht explizit im Code, sondern implizit – deklarativ – im Deployment Descriptor durch das Setzen eines Transaktionsattributs für die gesamte Bean (alle Methoden) oder einzelne Methoden. Die möglichen Transaktinsattribute und deren Auswirkungen auf die Methoden der EJBs sind in Tabelle 4.1 angeführt.

Attribut	**Beschreibung**
not supported	Transaktionen werden von der Methode, die dieses Attribut im Deployment Descriptor verwendet, nicht unterstützt. Besteht bereits beim Aufruf der Methode eine globale Transaktion, wird diese für die Ausführung der Methode unterbrochen (suspend) und nach Ende der Methode wieder fortgesetzt.
Required	Besteht beim Aufruf der Methode bereits eine globale Transaktion, so wird diese weiterverwendet. Besteht keine, wird eine neue Transaktion gestartet. Dieses Attribut gewährleistet also, dass die Methode stets innerhalb einer Transaktion ausgeführt wird.
Supports	Existiert bereits beim Methodenaufruf eine Transaktion, wird die Methode innerhalb dieses Transaktionskontexts ausgeführt. Existiert keine Transaktion, wird die Methode ohne Transaktion ausgeführt. Attribut eignet sich sehr gut für Methoden, die selbst keine Transaktion benötigen, aber an bereits begonnenen Transaktionen partizipieren sollen.
Requires New	Bei Verwendung dieses Attributs wird stets ein neuer Transaktionskontext erzeugt. Besteht bereits bei Aufruf der Methode eine Transaktion, wird diese angehalten und nach Beendigung der Methode (und der neuen Transaktion) wieder fortgesetzt. Die bereits bestehende Transaktion ist vom Methodenaufruf also nicht betroffen.
Mandatory	Wenn bereits eine globale Transaktion besteht, wird diese verwendet. Besteht keine, wird vom EJB-Container eine TransactionRequiredException geworfen. Das Attribut stellt sicher, dass die Methode immer die Transaktion des Clients verwenden muss.
Never	Wie der Name schon sagt, darf die dieses Attribut verwendende Methode nie innerhalb einer Transaktion aufgerufen werden. Besteht eine Transaktion, wird eine RemoteException geworfen, ansonsten wird die Methode ganz normal ausgeführt.

Tabelle 4.1: Transaktionsattribute J2EE

Listing 4.1 aus [SUN01] zeigt eine beispielhafte Festlegung von Transaktionsattributen im Deployment Descriptor. Alle Methoden der EJB EmployeeRecord werden mit dem Attribut „Required“ ausgeführt, lediglich für die Methode `updatePhoneNumber` wird das Attribut mit Mandatory überschrieben. Welche syntaktischen Möglichkeiten es zum Beschreiben der Container Managed Transaction gibt, wird in [SUN01] angeführt.

Listing 4.1:

```
<ejb-jar>
     ...
     <assembly-descriptor>
     ...
          <container-transaction>
               <method>
                    <ejb-name>EmployeeRecord</ejb-name>
                    <method-name>*</method-name>
               </method>
               <trans-attribute>Required</trans-attribute>
          </container-transaction>

          <container-transaction>
               <method>
                    <ejb-name>EmployeeRecord</ejb-name>
                    <method-name>updatePhoneNumber</method-name>
               </method>
               <trans-attribute>Mandatory</trans-attribute>
          </container-transaction>
          ...
     </assembly-descriptor>
</ejb-jar>
```

Da bei den Container Managed Transactions der EJB-Container die Transaktionssteuerung übernimmt, ist dieser auch für Rollbacks zuständig. Wird bei Ausführung einer Transaktion eine System Exception geworfen, wird vom Container automatisch ein Rollback durchgeführt. Möchte man „von Hand“ einen Rollback erzwingen, so muss man lediglich im Code einer (beteiligten) EJB die Methode `setRollbackOnly()` des Objekts `javax.ejb.EJBContext` aufrufen (Anmerkung: funktioniert nur bei Methoden, die die Attribute Required, RequiresNew oder Mandatory verwenden. Bei Supports, NotSupported oder Never wird bei Aufruf von `setRollbackOnly()` eine `java.lang.IllegalStateException` geworfen).

Um die Instanzvariablen einer stateful Session Bean mit ihren zugehörigen Werten in der Datenbank synchronisieren zu können, kann die Bean das Interface `javax.ejb.SessionSynchronization` – deren voids `afterBegin()`, `beforeCompletion()` und `afterCompletion(boolean committed)` zu den jeweiligen Zeitpunkten in der Transaktion vom Container aufgerufen werden –

implementieren. Auf die Synchronisation wird hier nicht weiter eingegangen, Details können unter anderem in [SUN01] nachgelesen werden.

Explizite Transaktionssteuerung

Neben der deklarativen Steuerung von Transaktionen gibt es in J2EE auch programmatische Möglichkeiten. Diese sind zum Einen Bean Managed Transactions und zum Anderen die Transaktionssteuerung mit Web Komponenten.

Bean Managed Transactions, die mit Session- und Message-driven- nicht aber bei Entity-Beans eingesetzt werden können, ermöglichen die explizite Verwendung von Transaktionen in den Methoden der Beans. Das verwendete Interface ist das bereits zuvor kurz erwähnte `javax.transaction.UserTransaction` Interface der Java Transaction API, das der EntwicklerIn die Möglichkeit des Starts (`UserTransaction.begin()`) und das Beenden (`UserTransaction.commit()` bzw. `UserTransaction.rollback()`) bietet. Während bei stateless Session-Beans und Message-driven Beans der Start und das Ende einer Transaktion innerhalb einer Methode passieren müssen (Transaktionen müssen beim Verlassen der Methode in der die Transaktion gestartet wurde beendet sein; bei Message-driven Beans vor Verlassen der `onMessage()` Methode), besteht bei stateful Session Beans die Möglichkeit, das Starten und Beenden globaler Transaktionen in verschiedenen Methoden durchzuführen. Dadurch ist es möglich, eine Transaktion über mehrere Requests aufrechtzuerhalten.

Alle – Ressource Manager betreffende – Aktionen zwischen `begin()` und `commit()` bzw. `rollback()` werden innerhalb einer globalen Transaktion ausgeführt. Zu beachten dabei ist zum Einen, dass während der globalen Transaktion keinerlei Transaktionsaufrufe über Ressource Manager ansprechende APIs gemacht werden dürfen (also bspw. kein Commit über eine `java.sql.Connection`) und zum Anderen nur eine Transaktion pro Thread laufen darf. Letztere Einschränkung bedeutet, dass vor Beendigung einer globalen Transaktion keine weitere gestartet werden darf, was wiederum bedeutet, dass zwar Interposition jedoch keine Schachtelung von Transaktionen möglich ist. Wird der Versuch dennoch unternommen, wird eine `java.lang.NotSupportedException` geworfen.

Neben den Bean Managed Transactions ist es wie bereits erwähnt auch möglich, unter Verwendung von JNDI globale Transaktionen mit Web Komponenten (Servlets, JSPs) zu steuern. Die Umsetzung erfolgt auch hier über das javax.transaction.UserTransaction Interface und bietet vor allem bei „fertigen" EJB-

Lösungen oder in Situationen, in denen auf die EJBs kein Einfluss genommen werden kann die Möglichkeit, trotzdem globale Transaktionen durchzuführen, wenngleich vor allem der Einsatz mit JSPs sehr kritisch zu betrachten ist (Stichwort MVC).

Implizite vs explizite Transaktionssteuerung

Nach der Vorstellung der beiden Transaktionssteuerungskonzepte stellt sich nun folgende Frage: „explizite oder implizite Steuerung?“, zu deren Beantwortung die nachstehenden Punkte helfen sollen:

- Performanceunterschiede zwischen impliziten und expliziten Transaktionen sind laut [Litt04] nicht sehr groß, da der Container bei expliziter Steuerung nach wie vor genügend Arbeit verrichten muss.
- Das Error-Handling ist bei Verwendung von impliziten Transaktionen wesentlich leichter
- Implizite Transaktionen werden normalerweise vom Hersteller eines J2EE-Produkts getestet (Korrektheit, Errorhandling,…)
- Durch den Einsatz von impliziten Transaktionen ist eine gute Trennung von Transaktionssteuerung und Anwendungslogik möglich.
- Die Wiederverwendbarkeit von Beans wird bei Verwendung deklarativer Transaktionen verbessert
- Explizite Transaktionen sind vom Client aus steuerbar und generell individuell besser anpassbar (kritisch dabei: bei der Implementierung können leicht Fehler passieren)
- Für Prototypen und kleine Aufgaben sind explizite Transaktionen aufgrund der raschen Implementierbarkeit im Regelfall besser geeignet

Eine explizite Transaktionssteuerung sollte daher gut überlegt werden und nur dann verwendet werden, wenn die Möglichkeit der deklarativen Steuerung nicht ausreicht. Dies wird auch in [SUN01] empfohlen.

Bei der Transaktionssteuerung ist es auch möglich, verschiedene Isolationslevel zu wählen und so Performancevorteile durch Erhöhung der Nebenläufigkeit zu erzielen. In dieser Hinsicht wird jedoch von Little et. al in [Litt04] zu großer Vorsicht geraten und empfohlen, „…fixe Isolationslevel für Ressource Adapter via Konfiguration zu setzen [Litt04]“. Wer dennoch mehr über Isolationslevel im Zusammenhang mit EJBs wissen möchte, kann die entsprechenden Informationen in [Litt04] sowie [SUN01] nachlesen.

4.1.2 Wichtiges zu Transaktionen mit J2EE

Zum Abschluss von Transaktionen mit EJB Komponenten und/oder verschiedenen Ressourcemanagern wird in J2EE der bereits in der Theorie vorgestellte, klassische Two-Phase Commit verwendet. Erstreckt sich die Transaktion lediglich über Komponenten eines J2EE Servers, wird der 2PC lokal durchgeführt, bei der Beteiligung von mehreren Servern ist dies ein verteilter 2PC (vgl. dazu verteilte Transaktionen, Abschnitt 2.5.1).

Das Ausführen von globalen Transaktionen über mehrere J2EE Server ist von der Architektur her grundsätzlich möglich und erfordert das Propagieren des Transaktionskontexts zu den beteiligten Komponenten. Es obliegt aber dem J2EE Plattform Provider, ob die Interoperabilität gewährleistet wird, oder nicht und ist natürlich auch sehr stark vom eingesetzten Transaktionsservice abhängig. In der J2EE 1.4 Spezifikation [SUN03] wird die Interoperabilität zwischen verschiedenen als auch gleichen J2EE Produkten freigestellt. Einzig wird für die Interoperabilität das IIOP transaction propagation protocol der OMG empfohlen.

Generell zu beachten bei globalen Transaktionen ist, dass diese wesentlich kostbarer und infolge dessen weniger performant als lokale Transaktionen sind. Der Einsatz von globalen Transaktionen sollte daher wohlüberlegt sein.

4.1.3 JDBC

Wie bereits in der Einleitung erwähnt, werden lokale Transaktionen direkt über den Ressourcemanager abgewickelt. Zu dieser Abwicklung stellt J2EE die Java Database Connectivity API (JDBC) als Schnittstelle zum Ressourcemanager zur Verfügung und bietet damit die Möglichkeit

- eine Verbindung zu relationalen Datenbanken herzustellen, diese
- via SQL anzusprechen (SQL Queries absetzen),
- Transaktionen zu starten und zu beenden sowie
- diverse Einstellungen vorzunehmen (AutoCommit, Isolationslevel)

Möchte man eine einfache Datenbankverbindung herstellen und einzelne SQL Statements (transaktional) ausführen, verwendet man in der Regel JDBC.

Werden Komponenten/Methoden, in denen lokale Operationen über JDBC ausgeführt werden innerhalb einer globalen Transaktion ausgeführt, so werden diese untereinander koordiniert. Im Klartext bedeutet diese, dass wenn bspw. zwei EJB-Methoden aufgerufen werden, von denen eine auf Datenbank x und y und die andere

auf Datenbank z zugreift, deren Ausgang mit einem 2PC untereinander abgestimmt wird. Dieser Sachverhalt ist in Abbildung 4.4 dargestellt.

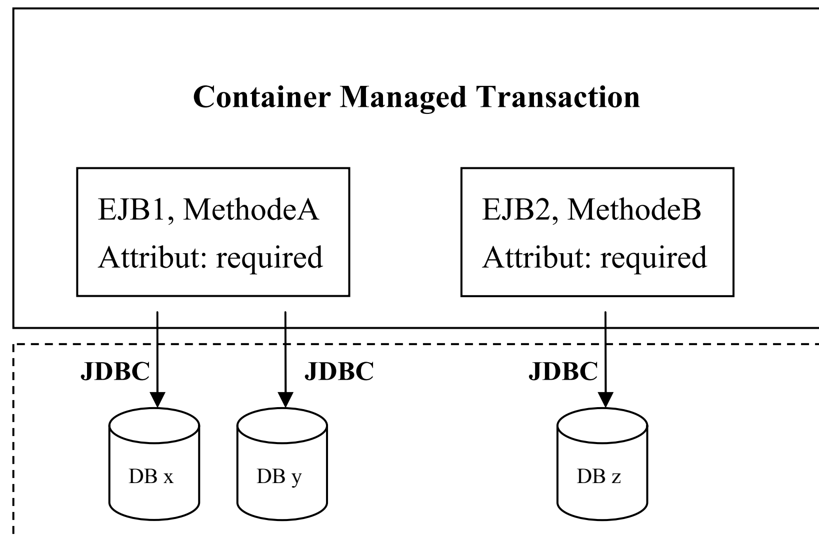


Abbildung 4.4: Container Managed Transaction und JDBC

4.1.4 J2EE und Holiday Package

Wie auf den vorherigen Seiten zu lesen, bietet J2EE durchaus umfangreiche Möglichkeiten für Transaktionen. Neben den lokalen Transaktionen und der damit verbundenen JDBC API für „klassische" Datenbankverbindungen sind vor allem globale Transaktionen, die eine Abstimmung unter mehreren Ressourcemanagern ermöglichen, für das Holiday Package interessant. So ist es beispielsweise möglich, Bean-Methoden für Flug-, Hotel- und Mietautobuchung innerhalb einer Transaktion auszuführen. Auch eine Verteilung dieser drei Teile auf drei verschiedene J2EE Server – und damit drei verschiedene Anbieter – ist möglich, wenngleich die Wahl des J2EE Produkts hier ganz entscheidend ist und über die Interoperabilitätsmöglichkeit bestimmt, wodurch eine Abstimmung/Einigung zwischen Anbietern und Servicenutzern unabdingbar ist. Für das Holiday Package ist dies natürlich eine große Einschränkung.

Der im Normalfall ausreichende und für die meisten transaktionalen Systeme sehr wichtige, in J2EE eingesetzte 2PC bedeutet eine klare Einschränkung der Nebenläufigkeit, da die Auswahl eines Holiday Packages oft sehr lange dauert und ein vorzeitiger Commit mit Kompensationsmöglichkeiten für die einzelnen Teile besser wäre. Auch die Unterscheidung zwischen vitalen und nicht vitalen Komponenten, wie sie

in der Theorie besprochen wurde, ist durch den 2PC nicht möglich, wie auch das Schachteln von Transaktionen in J2EE nicht erlaubt ist. Positiv zu erwähnen sind allerdings die verschiedenen Arten der Transaktionssteuerung, die eine Anpassung an individuelle Anforderungen ermöglichen.

Alles in Allem bietet J2EE im Grunde sehr gute Transaktionsmöglichkeiten, kann aber die Anforderungen, die das Holiday Package stellt – nämlich lange, über mehrere voneinander unabhängige heterogene Komponenten verteilte Transaktionen – nicht erfüllen.

Um das Holiday Package in J2EE dennoch adäquat durchführen zu können, bleibt allerdings die Möglichkeit, eines der drei in Kapitel 3 vorgestellten Transaktionsprotokolle für Web Services zu verwenden beziehungsweise zu implementieren. Ob beziehungsweise in welcher Form bestehende Lösungen/Produkte diese Möglichkeit unterstützen, ist Teil des Kapitels 5.

4.2 Transaktionen in .NET

Zur Erstellung der Inhalte dieses Unterkapitels wurden [Beer03] und [MICR05] herangezogen. Da Literatur für Transaktionen in .NET sehr schwer zu finden und eher dürftig ist, ist dieser Teil der Arbeit dementsprechend kurz ausgefallen.

Die in .Net zur Verfügung stehende Transaktionsverarbeitung unterscheidet lokale und verteilte Transaktionen. Während bei den lokalen nur eine Datenressource beteiligt ist, nehmen an der verteilten Transaktion mehrere Datenressourcen teil, um einen gemeinsamen Ausgang mit den ACID-Kriterien zu erzielen.

Eine Transaktion – durch einen transaction identifier gekennzeichnet – kann sich über Teile einer Methode bis hin zu ganzen Systemen erstrecken. Zur Festlegung dieser Transaktionsgrenzen stehen wie in J2EE zwei Modelle zur Verfügung: benutzerdefinierte Transaktionen (manual transactions) und automatische Transaktionen (automatic transactions)

Bei ersterer Variante erfolgt die Definition einer Transaktion explizit im Code und bietet die Möglichkeit der Schachtelung. Innerhalb der Grenzen einer Transaktion (parent-Transaktion) können also weitere Transaktionen (children) ausgeführt werden, was bedeutet, dass der parent mit dem Commit solange warten muss, bis alle seine Subtransaktionen mit Commit abgeschlossen sind.

Im Gegensatz zu den benutzerdefinierten Transaktionen erfolgt die Transaktionsdefinition bei automatischen Transaktionen deklarativ. Nesting wird von automatischen Transaktionen nicht unterstützt!

4.2.1 Automatische Transaktionen

Automatische Transaktionen, für deren Definition die Transaktionsdirektiven

- Disabled
- NotSupported
- Supported
- Required und
- RequiresNew

zur Verfügung stehen, werden je nachdem, ob es sich um eine ASP.NET Page, eine XML Web Service Methode oder um eine .NET Framework Klasse handelt, etwas anders definiert und auch die Auswirkung der Attribute auf das Verhalten der einzelnen Teile ist etwas unterschiedlich und in Tabelle 4.2 zusammengefasst.

Die Definition für eine ASP.NET Page erfolgt für die gesamte Seite und sieht folgendermaßen aus (in C#):

```
<% @Page Transaction="Required" %>
```

Bei XML Web Services erfolgt die Transaktionsdefinition separat für jede Methode direkt über dieser mittels folgender Deklaration (für C#):

```
[WebMethod (TransactionOption = TransactionOption.RequiresNew)]
```

Wichtig zu wissen bei der Definition des Transaktionsverhaltens von Web Service Methoden ist, dass durch die Tatsache, dass HTTP stateless ist (und Aufrufe von Web Services über HTTP-Requests erfolgen), jede Methode nur Wurzel (Root) einer Transaktion sein kann. Dies bedeutet aus praktischer Sicht, dass wenn Web Service y von Web Service x aufgerufen wird und sowohl für x als auch y Required oder RequiresNew als Attribut verwendet wird, y nicht innerhalb der Transaktion von x sondern innerhalb seiner eigenen Transaktion ausgeführt wird.

Für .Net Framework Klassen, die zur automatischen Transaktionsunterstützung von der ServicedComponent Klasse abgeleitet werden müssen (neben weiteren zusätzlich notwendigen Vorgängen; siehe [MICR05]), erfolgt die Deklaration des Transaktionsverhaltens direkt vor Beginn der Klasse durch folgenden Eintrag (in C#):

```
[Transaction (TransactionOption.Required)]
```

Attribut	ASP.NET	XML Web Service Methode	.NET Framework
Disabled	Ignorierung eines bestehenden Transaktionskontexts. (default)	Methode wird bei der Verarbeitung eine Requests ohne Transaktion ausgeführt.	Automatische Transaktionen sind deaktiviert.
NotSupported	Egal ob eine Transaktion aktiv ist oder nicht, läuft die Seite immer ohne Transaktion.	Siehe Disabled	Objekt läuft nie im Kontext einer Transaktion.
Supported	Existiert eine Transaktion, wird die Page innerhalb deren Kontexts ausgeführt, anderenfalls ohne.	Methode wird bei der Verarbeitung eines Requests ohne Transaktion erzeugt.	Wenn eine Transaktion existiert, erfolgt die Ausführung innerhalb dieser, anderenfalls ohne.
Required	Page wird jedenfalls innerhalb einer Transaktion ausgeführt. Existiert bereits eine, im Kontext dieser, ansonsten wird eine neue gestartet.	Methode benötigt Transaktion. Eine neue Transaktion wird erzeugt, da XML Web Service Methoden nur als Root an einer Transaktion teilnehmen können.	Objekt benötigt eine Transaktion. Besteht eine, erfolgt die Ausführung innerhalb dieser, anderenfalls wird eine neue erzeugt. (default)
RequiresNew	Für jeden Request wird eine neue Transaktion gestartet.	Methode wird innerhalb einer neuen Transaktion ausgeführt.	Objekt benötigt eine neue Transaktion.

Tabelle 4.2: Transaktionsattribute .NET

Voting

Um den Transaktionsabschluss in manchen Situationen beschleunigen zu können, stehen für .NET Framework Klassen und ASP.NET drei verschiedene Voting-Möglichkeiten, nämlich AutoComplete, SetAbort und SetComplete zur Verfügung.

Das Attribut AutoComplete, das in eckigen Klammern vor Methoden von der ServicedComponent Klasse abgeleiteten Klassen angegeben werden kann, bewirkt, dass bei Auftreten von Exceptions innerhalb des Methodenaufrufs das an der Transaktion teilnehmende Objekt Abort wählt und dadurch die gesamte Transaktion abgebrochen wird. Wird die Methode ohne Exception durchgeführt, wird ein Commit vorgeschlagen. Zur Definition des AutoComplete-Attributs sei noch gesagt, dass diese in der Klasse selbst erfolgen muss, erfolgt sie im Interface, wird sie ignoriert.

Die von der Klasse `System.EnterpriseServices.ContextUtil` zur Verfügung gestellten Methoden `SetComplete()` und `SetAbort()` können zur Signalisierung des vorzeitigen Commits beziehungsweise Aborts direkt im Code aufgerufen werden und so das Ende der Transaktion herbeizuführen.

Die Verwendung der vorgestellten Voting-Mechanismen ist insbesondere bei Auftreten von Fehlern interessant, da dadurch nicht das Ende der Transaktion abgewartet werden muss und die beteiligten Ressourcen nicht unnötig blockiert werden, sondern rasch wieder freigegeben werden können, was natürlich sehr große Performancevorteile mit sich bringen kann.

4.2.2 Benutzerdefinierte Transaktionen

Der Einsatz von benutzerdefinierten Transaktionen erlaubt es, die Transaktionsgrenzen selbst zu setzen. Durch Ausführung der Methode `BeginTransaction` des Objekts `Connection` - die `Connection` (z.B. `SqlConnection`) muss bereits geöffnet sein – wird eine lokale Transaktion gestartet und ein Transaktionsobjekt erzeugt, mittels dessen Methoden `Transaction.Commit()` beziehungsweise `Transaction.Rollback()` die Transaktion beendet werden kann.

Beim Ausführen der `BeginTransaction` Methode ist es durch Angabe einer von vier möglichen Isolationsstufen zusätzlich möglich, das Isolationsverhalten der Transaktion zu bestimmen.

Möchte man eine Connection zu einer Ressource innerhalb einer bestehenden, aktiven, verteilten Transaktion ausführen, so erfolgt die Registrierung (enlistment) au-

tomatisch, sofern das Auto-Enlistment, das durch Setzen von `Enlist=false` deaktiviert werden kann, aktiv ist. Ist das Auto-Enlistment deaktiviert, kann die Connection durch Aufruf der Methode `EnlistDistributedTransaction`, die als Übergabeparameter den Typ `ITransaction` – eine Referenz der Transaktion – benötigt, für die verteilte Transaktion registriert werden. Listing 4.2 (Auszug aus [MICR05], in C#) zeigt die Teilnahme beziehungsweise Registrierung für eine verteilte Transaktion.

Listing 4.2:

```
<%@Page Transaction="Required" %>
...
<html>
<Script Runat="SERVER" Language="C#">

void Page_Load()
{
  NorthwindSample ns = new NorthwindSample();
  ...
  ns.AddCustomer(customerId, companyName, (ITransac-
tion)ContextUtil.Transaction);
  ContextUtil.SetComplete();
  ...
}

public class NorthwindSample
{
   public void AddCustomer(string customerId, string company-
 Name, ITransaction trans)
  {
    SqlConnection nwindConn =  new SqlConnection("Data
       Source=localhost;Integrated Security=SSPI;" +
       "Initial Catalog=Northwind;Enlist=false;");

    SqlCommand cmd =
         new SqlCommand("INSERT INTO ...", nwindConn);

     cmd.Parameters.Add(...).Value = customerId;
     cmd.Parameters.Add(...).Value = companyName;

     nwindConn.Open();

     if (trans != null)
          nwindConn.EnlistDistributedTransaction(trans);
     ...
          cmd.ExecuteNonQuery();
     ...
          nwindConn.Close();
     ...
  }
}
</Script>
</html>
```

Wie bereits bei den Transaktionen in J2EE erwähnt, bringen benutzerdefinierte Transaktionen den Vorteil, die Transaktion flexibel gestalten zu können, doch bedeutet dies auch eine höhere Verantwortung für die ProgrammiererIn. Vor allem bei

verteilten Transaktionen, wo das Management von Recovery, Concurrency, Security und Integrity selbst übernommen werden muss (sinngemäß aus [Micr05]).

4.2.3 .NET und Holiday Package

Für .NET gilt in Bezug auf das Holiday Package das gleiche wie für J2EE, wenngleich aufgrund der sehr spärlich vorhandenen Informationen eine Analyse eher schwer fällt.

Für kurze (ACID) Transaktionen bietet .NET sowohl für lokale Transaktionen mit nur einer Ressource, als auch für verteilte Transaktionen mit mehreren Ressourcen adäquate Möglichkeiten der Durchführung. Auch mit der Möglichkeit der automatischen und benutzerdefinierten Transaktionen sind in .NET Mechanismen geboten, die eine individuelle und für verschiedenste Einsatzgebiete angemessene Transaktionssteuerung erlauben. Vor allem die benutzerdefinierten Transaktionen, die zwar einer größeren Verantwortung der Programmiererin bedürfen, bieten mit der Möglichkeit der Schachtelung interessantes Werkzeug zur Modularisierung und individuellen Anpassung.

Im Hinblick auf das Holiday Package Beispiel sind die kurzen Transaktionen und auch die Schachtelung, die – aufgrund der vorliegenden Informationen wird angenommen, dass es sich um eine geschlossene Schachtelung handelt (also kein vorzeitiges Sichtbarmachen von Teilergebnissen) – allerdings weniger geeignet.

Auch das Ausführen mehrerer Web Services innerhalb einer Transaktion ist nicht möglich, sodass der Einsatz von in Kapitel 3 vorgestellten Transaktionsprotokollen für eine angemessene Durchführung des Holiday Package Beispiels bleibt, und so die geforderte Interoperabilität zu anderen Systemen zu ermöglichen und auf verschiedene heterogene Systeme verteilte Dienste innerhalb einer langen, für Geschäftsprozesse angemessene Transaktionsart wie etwa WS-BA oder BPT durchführen zu können.

Kapitel 5

Implementierungen von Transaktionsprotokollen

Da nach den vorherigen Kapiteln Protokolle für die Durchführung verteilter Transaktionen mit Web Services vorgestellt wurden und bei den Transaktionen unter Einsatz von J2EE und .NET die Erkenntnis gemacht wurde, dass eine Problemstellung wie das Holiday Package am besten durch Implementierung eines der Web Service Transaktionsprotokolle umgesetzt werden können, stellt sich die Frage, in wieweit es bereits Implementierungen von diesen Protokollen gibt. Aus diesem Grund werden einige bereits existierende Produkte und Lösungen in diesem Kapitel kurz vorgestellt. Primär wird auf das Produkt „Arjuna Transaction Service Suite 4.0“ (ArjunaTS) von Arjuna Technologies eingegangen, weil dieses zum Einen sehr ausführlich dokumentiert ist und zum Anderen - aufgrund der Tatsache, dass es ein kommerzielles Produkt ist - eine gewisse Ausgereiftheit besitzt. Alle weiteren gefundenen Lösungen und Projekte werden nur kurz behandelt.

5.1 Arjuna Transaction Service Suite 4.0

Arjuna Technologies, die sich selbst auf ihrer Website wie folgt beschreiben:

„Arjuna Technologies is a leading independent supplier of distributed transaction processing and messaging software, helping companies guarantee business process reliability and safeguard mission-critical data.“ (www.arjuna.com)

, bieten Produkte im Bereich J2EE und Web Services an. Das für diese Arbeit sehr interessante Produkt „Arjuna Transaction Service Suite 4.0“ (ArjunaTS), herausgebracht im Februar 2005 ist eine Middleware, die die verteilte Transaktionsverarbeitung vereinfachen soll. Neben einem auf JTS basierenden – benutzt wird OTS – Transaktionsservice bietet ArjunaTS volle Unterstützung für verteilte Transaktionen (mit 2PC) sowie Recovery-Mechanismen und ist laut [Arju05] kompatibel mit den wichtigsten Standards. ArjunaTS ist 100% Java und somit sollte eine Installation auf verschiedensten Plattformen möglich sein. Eine genaue Beschreibung des Produkts kann in [Arju05] nachgelesen werden.

Der für diese Diplomarbeit interessanteste Teil von ArjunaTS ist zweifelsohne die Unterstützung von WS-Coordination, WS-AtomicTransaction und WS-BusinessActivity. Für die konkrete Implementierung dieser Protokolle stehen Java Interfaces, die auf dem JAXTX Standard (Java API for XML Transactioning) basieren, zur Verfügung. Diese bieten – gemäß den in Unterkapitel 3.3 vorgestellten Protokollen – entsprechende Methoden zum Registrieren für ein Protokoll und zum Setzen der entsprechenden Handlungen (z.B. Commit oder Befehl zu kompensieren). Eine genaue Beschreibung der API würde hier zu weit gehen. Eine entsprechende API Dokumentation ist im Produkt enthalten.

Die API wird in Zusammenhang mit Web Services eingesetzt. Die von ArjunaTS unterstützten Plattformen für Web Services sind:

- webMethods Glue 5 (www.webmethods.com)
- Apache Axis auf JBOSS (www.apache.org; www.jboss.org)
- BEA Web Logic (www.bea.com)

Um die Funktionsweise der API beziehungsweise deren Einsatz besser verstehen zu können, bietet Arjuna neben der eben genannten API Dokumentation und einem Programmer’s Guide ein sehr interessantes Demo, das im Wesentlichen den Sachverhalt des Holiday Package Beispiels widerspiegelt.

5.1.1 WS-C/WS-Tx Demo

Der im Demo herangezogene Sachverhalt zur Darstellung einer verteilten Transaktion ist die so genannte „Night Out“. Man möchte also an einem Abend in einem Restaurant essen und ins Theater gehen und anschließend mit dem Taxi nach Hause fahren. Diese drei Dinge sind mit Web Services zur Reservierung eines Tisches in einem Restaurant, der Reservierung von Theaterkarten sowie der Buchung eines

Taxis (die optional ist) realisiert. Die Transaktion kann wahlweise mit dem WS-AT oder dem WS-BA Protokoll durchgeführt werden.

Um das Demo zum Laufen zu bringen, müssen lediglich die Daten des Applikationsservers – zum Testen wurde JBoss 3.2.5 verwendet – wie etwa Pfad, URL und Port in einem property-File eingetragen werden. Durch die Ausführung eines Ant-Build-Scripts (Apache Ant, siehe www.apache.org) wird automatisch ein Projekt (in einer .ear-Archivdatei) im richtigen Verzeichnis des Applikationsservers erstellt. Ein Neustart des Servers genügt, um das Demo testen zu können. Der Screenshot in Abbildung 5.1 zeigt die Startseite des Demos, auf der die Art der Transaktion sowie die Details des „Night Out“ Pakets eingestellt werden können.

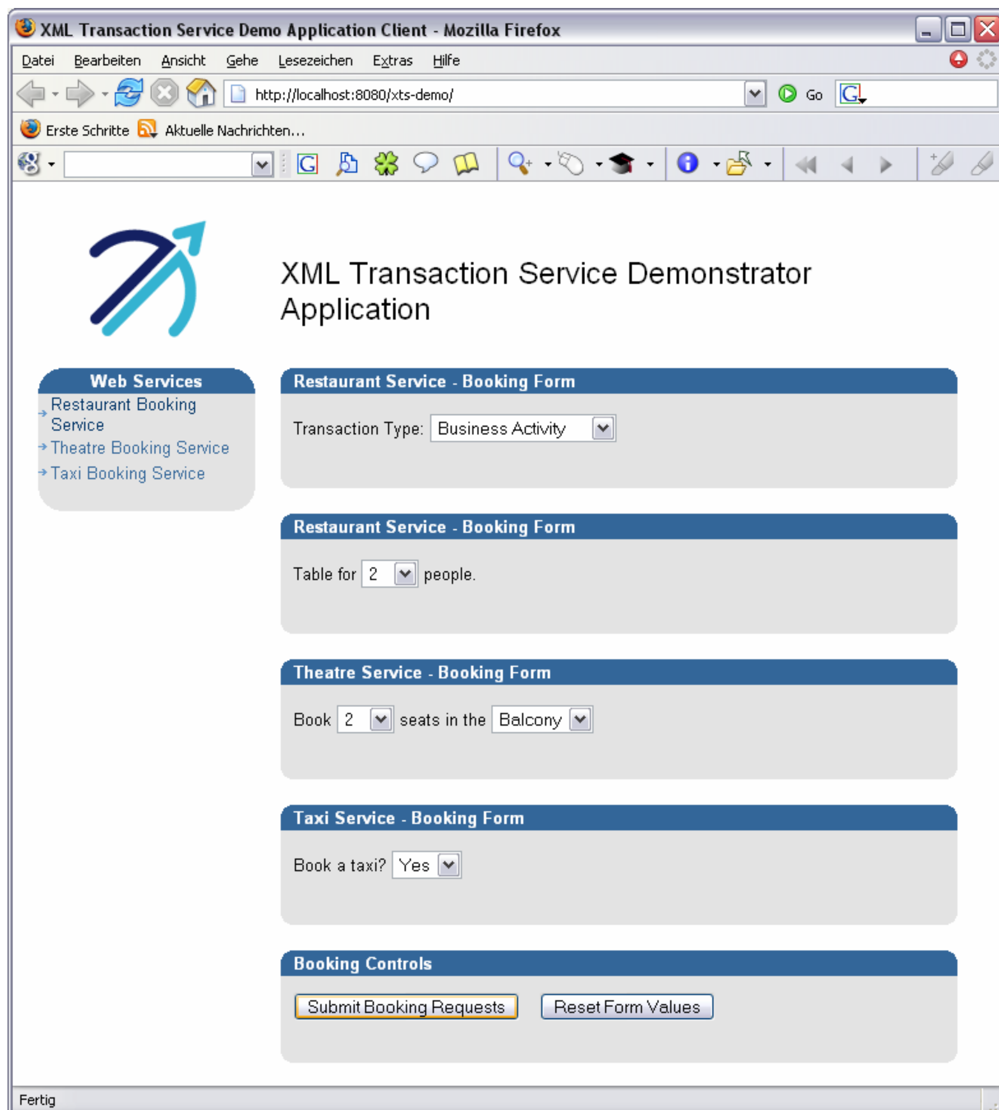


Abbildung 5.1: Startseite des ArjunaTS Demos

Durch Klick auf den Button „Submit Booking Request“ erfolgt die Buchung der einzelnen Komponenten innerhalb einer WS-AT/WS-BA Transaktion. Um die Reaktion der Web Services (mögliche Reaktionen vgl. Unterkapitel 3.3), wie etwa Fault, Exit, Completed etc. und deren Auswirkung auf alle Teilnehmer simulieren zu können, poppt beim Aufruf eines Services - also nach Klicken auf den Button „Submit Booking Request“ - für jedes Service ein Java-Fenster auf, mittels dem das Verhalten des Transaktionsteilnehmers gesteuert werden kann. In Abbildung 5.2 ist das Fenster für das Web Service Theater Reservierung beispielhaft dargestellt (verwendetes Protokoll: WS-BA).

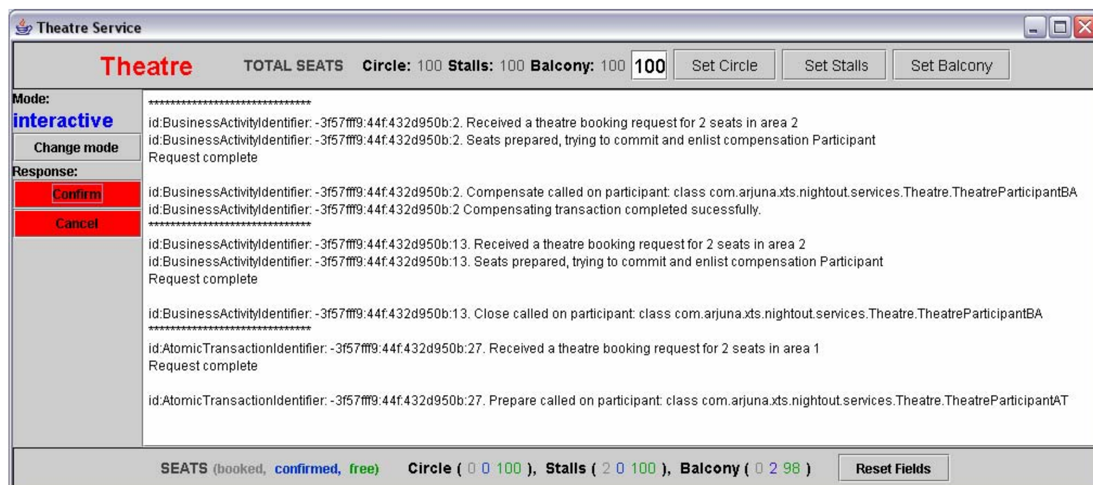


Abbildung 5.2: Kontrollfenster

Um zu testen, ob es möglich ist, nicht alle drei Services auf dem selben Server laufen zu lassen, sondern eines der Services von einem anderen Rechner aus in die Transaktion miteinzubeziehen, wurde das Demo auf einem zweiten Rechner deployed. Das Demo auf dem ersten Rechner wurde anschließend so konfiguriert, dass das Service nicht lokal, sondern über eine erreichbare URL aufgerufen wird (dies geschieht durch das Umändern des entsprechenden Eintrags im vorhin schon erwähnten property-File und das neue Deployen des Projekts). Der Versuch klappte wunderbar, das Service konnte vom zweiten Rechner aus an der Transaktion gestartet auf dem ersten Rechner teilnehmen und sein Verhalten während der Transaktion in bereits beschriebener Manier gesteuert werden (mittels Java-Fenster auf dem zweiten Rechner).

Der genaue technische Ablauf sowie die Erläuterung der genauen, konkreten Implementierung würde hier zu weit gehen. Der LeserIn wird daher ans Herz gelegt, sich bei Interesse ArjunaTS von http://www.arjuna.com herunterzuladen und zu installieren. Neben reichlich Dokumentation und der Java API können die gesamten Quelltexte des Demos eingesehen werden.

5.1.2 Fazit

Das Demo in ArjunaTS trifft eigentlich genau das Problem, das dieser Arbeit zugrunde liegt und stellt einen durchaus zufrieden stellenden Lösungsansatz dar, der durch die angebotene API den für eine rasche und unkomplizierte Implementierung nötigen Abstraktionsgrad bietet.

Ob die angebotene Unterstützung von WS-C/WS-Tx auch in Kombination mit Produkten anderer Hersteller funktioniert, müsste erst getestet werden. Zu beachten bei ArjunaTS ist auch, dass es sich dabei um ein kommerzielles und kein Open Source Produkt handelt, das zwar zur Evaluierung gratis getestet werden kann, bei professionellem Einsatz allerdings kostenpflichtig ist (Preise abhängig von der Anzahl der CPUs des Rechners auf dem ArjunaTS läuft (einmalige Zahlung) sowie jährliche Supportkosten). Allerdings bedeutet ein kommerzielles Produkt auch einen gewissen Druck beim Anbieter, dass die angebotenen Produkte auch funktionieren müssen, um erfolgreich verkauft werden zu können.

5.2 JOTM-BTP

Java Open Transaction Manager (JOTM) ist - wie der Name vermuten lässt - ein Open Source Transaktionsmanger, implementiert in Java. JOTM bietet zwar noch keinen vollen Support von OTS, doch ist dieser geplant. Neben diesem sind laut [OBJE05] auch Prototypen von offenen und geschlossenen geschachtelten Transaktionen sowie die Interoperabilität zwischen verschiedenen Transaktionsmodellen und Spezifikationen geplant, was eine Weiterverfolgung der Entwicklung von JOTM sehr interessant macht.

Zu JOTM gibt es eine BTP Erweiterung mit dem Namen JOTM-BTP. Diese Erweiterung erfordert den Einsatz von Tomcat (www.apache.org), Ant 1.5.x, Axis 1.0 sowie JOTM 1.4. Für die BTP Erweiterung wird ähnlich wie bei ArjunaTS ein Demo, nämlich ein Holiday Package mit drei Web Services – Travel Agent Web Service bucht Hotel und Flug (jeweils als Web Service zur Verfügung gestellt) - zum Testen von BTP zur Verfügung gestellt.

Die BTP Implementierung unterstützt keine Cohesions, nur Atoms, jedoch ist es laut [MESN03] durch einen „Hack“ möglich, Cohesions doch zu unterstützen.

JOTM-BTP wurde nicht im Detail analysiert. Die interessierte LeserIn wird deshalb auf [MESN03] verwiesen, worin sowohl eine Installationsanleitung für die Erweite-

rung und das Deployment des Demos als auch Source-Codes des Demos enthalten sind. Weitere Informationen zu JOTM und JOTM-BTP sind unter http://jotm.objectweb.org/ verfügbar.

5.3 Apache Kandula

Kandula ist ein Projekt von Apache, das sich unter anderem eine Open-Source Implementierung von WS-Coordination, WS-AtomicTransaction und WS-BusinessActivity basierend auf Axis als Ziel gesetzt hat. Auch die Interoperabilität zu anderen Implementierungen der Spezifikationen von WS-C/WS-Tx (siehe Unterkapitel 3.3) wird angestrebt (nach http://ws.apache.org/kandula).

Bisher wurden WS-C und WS-AT implementiert, eine Umsetzung von WS-BA soll allerdings folgen. Informationen zu diesem Projekt wie etwa aktueller Status, Design der Architektur, Ziele oder Quelltexte können unter http://ws.apache.org/kandula erhalten werden.

5.4 Weitere Implementierungen

Weitere Projekte, die sich mit Transaktionen für Web Services beschäftigen, die in dieser Arbeit allerdings nicht behandelt werden, sind:

- WS-AT for WebSphere Application Server (WS-AT for WAS) (http://www.alphaworks.ibm.com/tech/wsat)
- Emerging Technologies Toolkit for Web Services and Autonomic Computing von IBM (http://www.alphaworks.ibm.com/tech/ettkws)

Kapitel 6

Zusammenfassung und Ausblick

In dieser Arbeit wurde versucht, die Möglichkeiten, über mehrere (heterogene) Systeme verteilte Transaktionen durchzurühren, zu erörtern. Im Vordergrund standen dabei nicht kurze Transaktionen, sondern verteilte, lange Transaktionen, mittels deren Einsatz auch komplexen Geschäftsprozessen Rechnung getragen werden kann. Als begleitendes Beispiel für einen derartigen Prozess wurde das Holiday Package herangezogen und die in dieser Arbeit vorgenommene Evaluierung anhand dieses Beispiels durchgeführt.

Zuerst wurde auf Transaktionen und einige wichtige Aspekte im Allgemeinen eingegangen sowie einige für lange, verteilte Transaktionen in Frage kommende Transaktionsarten vorgestellt. Dabei zeigte sich, dass vor allem offene geschachtelte Transaktionen besonders für komplexe Sachverhalte, bei denen auch Nebenläufigkeit und Performanz wichtig sind, geeignet sind.

Da im Moment Web Services DER Standard für die Kommunikation von verschiedenen heterogenen Systemen sind, wurden in Kapitel 3 aktuelle Ansätze zur Durchführung von Transaktionen mit Web Services untersucht und gegenübergestellt. Dabei zeigte sich, dass alle drei herangezogenen Protokolle (BTP, WS-C/WS-Tx, WS-CAF) im Grunde genommen sehr brauchbar für die Durchführung des Holiday Package Beispiels – also für lange Transaktionen – sind. Natürlich unterscheiden sich die drei Protokolle vom Konzept her teilweise sehr stark, was sich in den Möglichkeiten und der Handhabbarkeit der einzelnen Protokolle niederschlägt, wie die durchgeführte Gegenüberstellung der Protokolle zeigt.

Im vierten Kapitel wurde versucht, die Transaktionsmöglichkeiten der Frameworks J2EE und .NET zu beleuchten, da diese zum Einen in der Praxis sehr häufig eingesetzt werden und zum Anderen Web Services oftmals in dieser Umgebung eingebettet sind. Dabei konnte festgestellt werden, dass es für kurze Transaktionen, die den ACID-Kriterien genügen müssen, adäquate Möglichkeiten gibt. Für lange Transaktionen über mehrere (heterogene) Systeme ist die Eignung hingegen nicht gegeben.

Am Ende dieser Arbeit wurden einige Projekte und Ansätze vorgestellt, die sich mit der Umsetzung der im dritten Kapitel vorgestellten Transaktionsprotokolle beschäftigen. Dabei wurde vor allem auf ArjunaTS von Arjuna Technologies eingegangen, das die Protokollkombination WS-C/WS-Tx in Form einer Java API zur Verfügung stellt mit einem mitgelieferten Demo zeigt, wie ein Beispiel ähnlich dem des Holiday Packages mit Web Services und langen Transaktionen implementiert und umgesetzt werden kann.

Alles in Allem ist zu sagen, dass es sehr gute Ansätze in Richtung lange, verteilte Transaktionen gibt. Beim Recherchieren und Ausarbeiten hatte ich das Gefühl, dass dieses Gebiet keines ist, in dem kaum Forschung betrieben sondern es sehr viel Interesse und Arbeiten in diese Richtung gibt, was nicht zuletzt die ständigen Weiterentwicklungen von Protokoll-Spezifikationen und Ziele von entstandenen Projekten zeigen.

Gerade deshalb gibt es noch einige offene Punkte und Ideen, die den Rahmen dieser Diplomarbeit bei weitem gesprengt hätten, aber dennoch interessant für weitere Arbeiten wären:

- Wie vorgestellt, gibt es verschiedene Projekte, die sich mit der Implementierung von Transaktionsprotokollen auseinandersetzen. Da die Umsetzung erst teilweise passiert ist, wäre es interessant, diese weiterzuverfolgen und diese in weiterer Form vor allem hinsichtlich ihrer Kompatibilität untereinander zu testen.
- Weiters sind viele Spezifikationen von Transaktionsprotokollen gerade im Umbruch. So gibt es etwa zu WS-AtomicTransaction und WS-BusinessAcitvity neue Versionen vom August 2005. Auch WS-CAF wird gerade überarbeitet. Dazu wäre die Frage zu beantworten, was sich geändert hat/ändern wird und welche positiven und negativen Auwirkungen die Änderungen auf konkrete Problemstellungen haben.
- Da nicht gesagt werden kann, ob beziehungsweise welcher Ansatz sich durchsetzen wird, könnte versucht werden, geeignete Regeln zu entwickeln

und zu definieren, um eine Interoperabilität unter den Protokollen herzustellen.

- Für J2EE gibt es einen Ansatz, der sich mit der Durchführung langer Transaktionen befasst, der allerdings nicht behandelt werden konnte, dessen Möglichkeiten und der aktuelle Status aber durchaus von Interesse wären (Stichwort J2EE Activity Service, siehe [Litt04]).
- Das Betrachten von Transaktionen auf anderer Ebene etwa der Modellierungsebene wäre ein weiterer Punkt. Zur expliziten Darstellung von Transaktionen bei der Modellierung von Geschäftsprozessen gibt es kaum Möglichkeiten. Bisherige Ansätze zu evaluieren oder neue zu entwickeln und eventuell das Mapping zu konkreten Transaktionsprotokollen herzustellen wäre ein weiterer interessanter Aspekt.

Abbildungsverzeichnis

Abbildung 2.1: Ausführung von Operationen mit und ohne Transaktionskontext...... 6
Abbildung 2.2: Aufbau einer Transaktion 7
Abbildung 2.3: Konten einer Bank (nach [Bern87]).. 11
Abbildung 2.4: Konfliktrelationen zwischen Lese- und Schreiboperationen (nach [Elma92]) 14
Abbildung 2.5: Zusammenhang zwischen Serialisierungstypen 15
Abbildung 2.6: Two-Phase Locking (nach [Litt04]).. 18
Abbildung 2.7: Verteilte Transaktion (nach [Litt04]).. 23
Abbildung 2.8: Phasen eines Two-Phase Commit (nach [Litt04]) 24
Abbildung 2.9: Beispiel einer geschachtelten Transaktion.. 28
Abbildung 3.1: Überblick über den Einsatz von Web Services (aus [Rahm03])....... 33
Abbildung 3.2: Aufbau einer SOAP-Nachricht (aus [Rahm03]) 34
Abbildung 3.3: Web Services, Transaktionen und Context (aus [Litt04]) 36
Abbildung 3.4: Superior:Inferior Beziehung (aus [Furn04]) 39
Abbildung 3.5: Transaktion mit einem Intermediate (interposition) - (nach [Furn04]) 40
Abbildung 3.6: BTP Multi Service Type Cohesion (aus [Pott02])............................ 45
Abbildung 3.7: Beispielhafter Ablauf von Aktivierung und Registrierung in WS-Coordination... 49
Abbildung 3.8: WS-AT Completion Protokoll (aus [Cabr04b])................................ 50
Abbildung 3.9: Two-Phase Commit Protokoll in WS-AT (aus [Cabr04b]) 51
Abbildung 3.10: BusinessAgreementWithParticipantCompletion Protokoll 53
Abbildung 3.11: BusinessAgreementWithCoordinatorCompletion Protokoll 54
Abbildung 3.12: Schichten von WS-CAF.. 59
Abbildung 3.13: Zusammenhang zwischen WS-CTX, WS-CF und WS-TXM (aus [Bunt03d]) 60
Abbildung 3.14: Ablauf des synchronization-Protokolls... 61
Abbildung 3.15: Beispielhafter Ablauf einer LRA 63
Abbildung 3.16: Business Processes und Tasks 65
Abbildung 3.17: Statuswechsel eines Business Processes (aus [Bunt03c])............... 66
Abbildung 3.18: terminate-notification Protokoll (aus [Bunt03c]) 67

Abbildung 3.19: businessProcess Protokoll (aus WS-TXM)....................................67
Abbildung 3.20: checkpoint Protokoll (aus [Bunt03c])..68
Abbildung 3.21: restart Protokoll (aus [Bunt03c]) 69
Abbildung 3.22: workStatus Protokoll (aus [Bunt03c]) 69
Abbildung 3.23: completion Protokoll (aus [OASI05])..70
Abbildung 3.24: Kombination von LRAs..71
Abbildung 4.1: Zusammenspiel der Komponenten 77
Abbildung 4.2: Zusammenhang der Komponenten (aus [Sun99]) 78
Abbildung 4.3: Interaktion der Objekttypen (aus [OMG01])....................................80
Abbildung 4.4: Container Managed Transaction und JDBC 86
Abbildung 5.1: Startseite des ArjunaTS Demos 95
Abbildung 5.2: Kontrollfenster 96

Tabellenverzeichnis

Tabelle 3.1: Einige Alternativen für provisorische, endgültige und Gegeneffekte ... 38
Tabelle 3.2: BTP und ACID (nach [Pott02]) ... 42
Tabelle 3.3: Gegenüberstellung der Protokolle ... 75
Tabelle 4.1: Transaktionsattribute J2EE ... 81
Tabelle 4.2: Transaktionsattribute .NET ... 89

Literaturverzeichnis

[Arju05] Arjuna Technologies: Arjuna Transaction Service. http://www.arjuna.com/library/product/arjunats/2005-02-ArjunaTS-datasheet.pdf (27.9.2005), 2005

[Beer03] W. Beer, D. Birngruber, H. Mössenböck, A. Wöß: Die .NET-Technologie. dpunkt.verlag, Heidelberg, 2003

[Bern87] P.A. Bernstein, V. Hadzilacos, N. Goodman: Concurrency Control and Recovery in Database Systems. Addison-Wesley, Reading, Mass. [u.a.], 1987

[Box04] D. Box et al.: Web Services Addressing (WS-Addresing). http://www.w3.org/Submission/2004/SUBM-ws-addressing-20040810/ (20.9.2005), 2004

[Bunt03a] D. Bunting et al.: Web Services Context Service Specification. http://developers.sun.com/techtopics/webservices/wscaf/wsctx.pdf (20.9.2005), 2003

[Bunt03b] D. Bunting et al.: Web Services Coordination Framework Specification. http://developers.sun.com/techtopics/webservices/wscaf/wscf.pdf (20.9.2005), 2003

[Bunt03c] D. Bunting et al.: Web Services Transaction Management Specification. http://developers.sun.com/techtopics/webservices/wscaf/wstxm.pdf (20.9.2005), 2003

[Bunt03d] D. Bunting et al.: Web Services Composite Application Framework. http://developers.sun.com/techtopics/webservices/wscaf/primer.pdf (20.9.2005), 2003

[Cabr04a] Cabrera et al.: Web Services Coordination (WS-Coordination). http://xml.coverpages.org/WS-Coordination200411.pdf (20.9.2005), 2004

[Cabr04b] Cabrera et al.: Web Services Atomic Transaction (WS-AtomicTransaction). http://xml.coverpages.org/WS-AtomicTransaction200411.pdf (20.9.2005), 2004

[Cabr04c] Cabrera et al.: Web Services Business Activity Framework (WS-BusinessActivity). http://xml.coverpages.org/WS-BusinessActivity200411.pdf (20.9.2005), 2004

[Elma92] A.K. Elmagarmid: Database Transaction Models for Advanced Applications. Morgan Kaufmann Publishers, San Mateo, California, 1992

[Elma02] R. Elmasri, S.B. Navathe: Grundlagen von Datenbanksystemen. 3., überarbeitete Auflage. Pearson Studium, München, 2002

[Furn04] P. Furniss et al.: OASIS Business Transaction Protocol, Version 1.0.9.5 - BTP 1.1 Working Draft 05. http://xml.coverpages.org/BTPv11-200411.pdf (20.9.2005), 2004

[Gray93] J. Gray, A. Reuter: Transaction Processing - Concepts and Techniques, Morgan Kaufmann Publishers, San Francisco, California, 1993

[Litt03] M. Little, T. Freund: A comparison of Web services transaction protocols – A comparative analysis of WS-C/WS-Tx and OASIS BTP. http://www-128.ibm.com/developerworks/webservices/library/ws-comproto/ (20.9.2005), 2003

[Litt04] M. Little, J. Maron, G. Pavlik: Java Transaction Processing : Design and Implementation. Prentice Hall PRT, Upper Saddle River, New Jersey, 2004

[Mesn03] J. Mesnil: JOTM-BTP: a BTP extension for JOTM. http://jotm.objectweb.org/current/jotm-btp/doc/btp.pdf (27.9.2005), 2003

[Micr05] Microsoft Developer Network: Processing Transactions. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconProcessingTransactions.asp (27.9.2005), 2005

[OASI05] OASIS Consortium: Web Services Business Process Specification. http://www.oasis-open.org/committees/download.php/12793/WS-BP.ZIP (20.9.2005), 2005

[OBJE05] ObjectWeb Consortium: Java Open Transaction Manager. http://jotm.objectweb.org/doc/jotm.pdf (27.9.2005), 2005

[OMG01] Object Management Group: Object Transaction Service Specification, Version 1.2. http://www.omg.org/docs/formal/01-11-03.pdf (27.9.2005), 2001

[Öszu99] M.T. Öszu, P. Valduriez: Principles of Distributed Database Systems. Second Edition, Prentice Hall, Upper Saddle River, New Jersey, 1999

[Pott02] M. Potts et al.: Business Transaction Protocol Primer. http://www.oasis-open.org/committees/download.php/2077/BTP_Primer_v1.0.20020605.pdf (20.9.2005), 2002

[Rahm03] E. Rahm, G. Vossen (Hrsg.): Web & Datenbanken - Konzepte, Architekturen, Anwendungen, dpunkt.verlag, Heidelberg, 2003

[Sun99] Sun Microsystems: Java Transaction Service Specification, Version 1.0. http://java.sun.com/products/jts/ (27.9.2005), 1999

[Sun01] Sun Microsystems: Enterprise JavaBeans Specification, Version 2.0. http://java.sun.com/products/ejb/docs.html (27.9.2005), 2001

[Sun02] Sun Microsystems: Java Transaction API Specification, Version 1.0.1 B. http://java.sun.com/products/jta/ (27.9.2005), 2002

[Sun03] Sun Microsystems: Java 2 Platform Enterprise Edition Specification, Version 1.4. http://java.sun.com/j2ee/j2ee-1_4-fr-spec.pdf (27.9.2005), 2003